目錄

计算与推断思维 中文版

原书：data-8/textbook

译者：飞龙

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- 代码仓库

赞助我



协议

KivyCN 学习资源

- Think Python 中文第二版
- UCB CS61a 教材：SICP Python
- Tutorialspoint NumPy 教程
- Matplotlib 用户指南
- 斯坦福 CS229 机器学习中文讲义
- Duke STA663 计算统计学中文讲义
- 笨办法学 Python · 续

一、数据科学

> 原文：Data Science
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

什么是数据科学

数据科学是通过探索，预测和推断，从大量不同的数据集中得出有用的结论。探索涉及识别信息中的规律。预测涉及使用我们所知道的信息，对我们希望知道的值作出知情的猜测。推断涉及量化我们的确定程度：我们发现的这些规律是否也出现在新的观察中？我们的预测有多准确？我们用于探索的主要工具是可视化和描述性统计，用于预测的是机器学习和优化，用于推理的是统计测试和模型。

统计学是数据科学的核心部分，因为统计学研究，如何用不完整的信息做出有力的结论。计算是一个重要组成部分，因为编程允许我们将分析技巧应用于大量不同的数据集，它们在真实应用中出现：不仅包括数字，还包括文本，图像，视频和传感器读数。数据科学就是所有这些东西，但是由于应用的原因，它不仅仅是其部分的总和。通过理解一个特定的领域，数据科学家学习提出有关他们的数据的适当的问题，并正确地解释我们的推理和计算工具提供的答案。

简介

数据是对我们周围世界的描述，通过观察来收集并存储在计算机上。计算机使我们能够从这些描述中推断出世界的特性。数据科学是使用计算从数据中得出结论的学科。有效的数据分析有三个核心方面：探索，预测和推理。本文对三者进行了一致的研究，同时介绍了统计思想和计算机科学的基本思想。我们专注于一套最小的核心技巧，应用于广泛的实际应用。数据科学的基础不仅需要理解统计和计算技巧，还需要认识到它们如何应用于真实场景。

对于我们希望研究的世界的任何方面，无论是地球气象，世界市场，政治民意调查还是人类思想，我们收集的数据通常都提供了这个主题的不完整描述。数据科学的核心挑战是使用这部分信息作出可靠的结论。

在这个努力中，我们将结合两个基本工具：计算和随机化。例如，我们可能想使用温度观测来了解气候变化的趋势。计算机允许我们使用所有可用的信息得出结论。我们不仅仅关注一个地区的平均气温，而是将整个温度的范围一起考虑，来构建更加细致的分析。随机性允许

我们考虑许多不同方式，来完善不完整的信息。我们不会假设温度会以某种特定的方式变化，而是学习使用随机性来设想许多可能的情景，这些情景都与我们观察到的数据一致。

应用这种方法需要学习，如何为一台计算机编程，所以这个文本穿插了编程的完整介绍，并假设没有任何先验知识。 具有编程经验的读者会发现，我们涵盖了计算中的几个主题，这些主题并没有出现在典型的计算机科学课程中。 数据科学也需要对数量进行仔细的推理，但是本书并不假设超出基本代数的数学或统计背景。 在本文中你会发现很少的方程。 相反，技巧使用一种编程语言描述，对于读者和执行它们的计算机来说，是相同的。

计算工具

本文使用 Python 3 编程语言，以及数值和数据可视化的标准工具集，它们在商业应用，科学实验和开源项目中广泛使用。 Python 已经招募了许多专业人士，它们使用数据得出结论。通过学习 Python 语言，你将加入一个拥有百万人口的，软件开发人员和数据科学家社区。

入门。开始用 Python 编写程序的最简单和推荐的方法是，登录到本文的配套网站 https://datahub.berkeley.edu/。如果你拥有 `@ berkeley.edu` 电子邮件地址，则你已经可以完全访问该网站上托管的编程环境。如果没有，请填写此表格来申请访问。

你不能完全仅仅使用这个基于 Web 的编程环境。 Python 程序可以由任何计算机执行，无论其制造商或操作系统如何，只要安装了该语言的支持。如果你希望安装符合本文的 Python 版本及其附带库，我们推荐将 Anaconda 发行版与 Python 3 语言解释器，IPython 库和 Jupyter 笔记本环境打包在一起。

本文包括所有这些计算工具的完整介绍。你将学习编写程序，从数据生成图像，并使用在线发布的真实世界的数据集。

统计技巧

统计学科长期以来一直面临与数据科学相同的根本挑战：如何使用不完整的信息得出有关世界的有力结论。统计学最重要的贡献之一是，用于描述观察与结论之间关系的，一致而准确的词汇。本文继续保持同样的传统，重点是统计学中的一组核心推断问题：假设检验，置信度估计和未知量预测。

数据科学通过充分利用计算，数据可视化，机器学习，优化和信息访问来扩展统计领域。快速计算机和互联网的结合使得任何人都能够访问和分析大量的数据集：数百万篇新闻文章，完整的百科全书，任何领域的数据库以及大量的音乐，照片和视频库。

真实数据集的应用激发了我们在整个文本中描述的统计技巧。真实数据通常没有规律或匹配标准方程。如果把过多的注意力集中在简单的总结上，比如平均值，那么真实数据中有趣的变化就会丢失。计算机使一系列基于重采样的方法成为可能，它们适用于各种不同的推理问题，考虑了所有可用的信息，并且需要很少的假设或条件。虽然这些技巧经常留作统计学的研究生课程，但它们的灵活性和简单性非常适合数据科学应用。

为什么是数据科学

最重要的决策仅仅使用部分信息和不确定的结果做出。然而，许多决策的不确定性，可以通过获取大量公开的数据集和有效分析所需的计算工具，而大幅度降低。以数据为导向的决策已经改变了一大批行业，包括金融，广告，制造业和房地产。同时，大量的学科正在迅速发展，将大规模的数据分析纳入其理论和实践。

学习数据科学使个人能够将这些技巧用于工作，科学研究和个人决策。批判性思维一直是严格教育的标志，但在数据支持下，批判往往是最有效的。对世界任何方面的批判性分析，可能是商业或社会科学，涉及归纳推理；结论很少直接证明，仅仅由现有的证据支持。数据科学提供了手段，对任何一组观测结果进行精确，可靠和定量的论证。有了信息和计算机的前所未有的访问，如果没有有效的推理技巧，对世界上任何可以衡量的方面的批判性思考都是不完整的。

世界上有太多没有答案的问题和困难的挑战，所以不能把这个批判性的推理留给少数专家。所有受过教育的社会成员都可以建立推断数据的能力。这些工具，技巧和数据集都是随手可用的；本文的目的是使所有人都能访问它们。

绘制经典作品

在这个例子中，我们将探讨两个经典小说的统计：马克吐温（Mark Twain）的《哈克贝利·芬历险记》（The Adventures of Huckleberry Finn）和路易莎·梅·奥尔科特（Louisa May Alcott）的《小女人》（Little Women）。任何一本书的文本都可以通过电脑以极快的速度读取。 1923 年以前出版的书籍目前处于公有领域，这意味着每个人都有权以任何方式复制或使用文本。 古登堡计划是一个在线出版公共领域书籍的网站。使用 Python，我们可以直接从网络上加载这些书籍的文本。

这个例子是为了说明本书的一些广泛的主题。如果还不理解程序的细节，别担心。相反，重点关注下面生成的图像。后面的部分将介绍下面使用的 Python 编程语言的大部分功能。

首先，我们将这两本书的内容读入章节列表中，称为`huck_finn_chapters`和`little_women_chapters`。在 Python 中，名称不能包含任何空格，所以我们经常使用下划线`_`来代表空格。在下面的行中，左侧提供了一个名称，右侧描述了一些计算的结果。统一资源定位符或 URL 是互联网上某些内容的地址；这里是一本书的文字。`#`符号是注释的起始，计算机会忽略它，但有助于人们阅读代码。

```
# Read two books, fast!

huck_finn_url = 'https://www.inferentialthinking.com/chapters/01/3/huck_finn.txt'
huck_finn_text = read_url(huck_finn_url)
huck_finn_chapters = huck_finn_text.split('CHAPTER ')[44:]

little_women_url = 'https://www.inferentialthinking.com/chapters/01/3/little_women.txt'

little_women_text = read_url(little_women_url)
little_women_chapters = little_women_text.split('CHAPTER ')[1:]
```

虽然计算机不能理解书的文本，它可以向我们提供文本结构的一些视角。名称 `huck_finn_chapters` 现在已经绑定到书中章节的列表。我们可以将其放到一个表中，来观察每一章的开头。

```
# Display the chapters of Huckleberry Finn in a table.

Table().with_column('Chapters', huck_finn_chapters)
```

Chapters
I. YOU don't know about me without you have read a book ...
II. WE went tiptoeing along a path amongst the trees bac ...
III. WELL, I got a good going-over in the morning from o ...
IV. WELL, three or four months run along, and it was wel ...
V. I had shut the door to. Then I turned around and ther ...
VI. WELL, pretty soon the old man was up and around agai ...
VII. "GIT up! What you 'bout?" I opened my eyes and look ...
VIII. THE sun was up so high when I waked that I judged ...
IX. I wanted to go and look at a place right about the m ...
X. AFTER breakfast I wanted to talk about the dead man a ...

（已省略 33 行）

每一章都以章节号开头，以罗马数字的形式，后面是本章的第一个句子。古登堡计划将每章的第一个单词变为大写。

文本特征

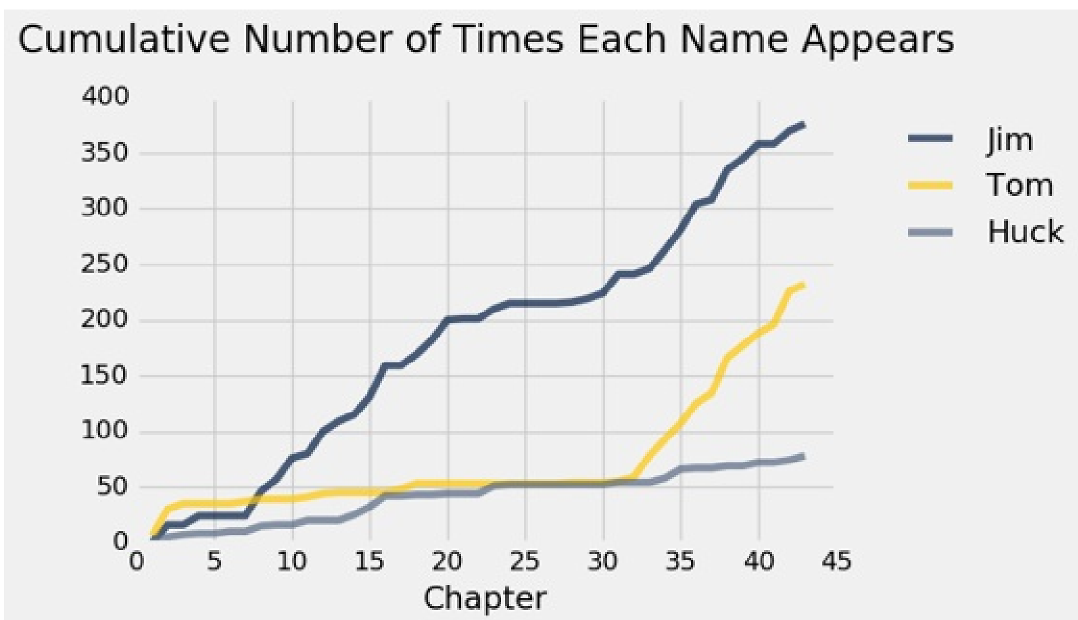
《哈克贝利·芬历险记》描述了哈克和吉姆沿着密西西比河的旅程。汤姆·索亚（Tom Sawyer）在行动进行的时候加入了他们的行列。在加载文本后，我们可以快速地看到这些字符在本书的任何一处被提及的次数。

```
# Count how many times the names Jim, Tom, and Huck appear in each chapter.

counts = Table().with_columns([
        'Jim', np.char.count(huck_finn_chapters, 'Jim'),
        'Tom', np.char.count(huck_finn_chapters, 'Tom'),
        'Huck', np.char.count(huck_finn_chapters, 'Huck')
    ])

# Plot the cumulative counts:
# how many times in Chapter 1, how many times in Chapters 1 and 2, and so on.

cum_counts = counts.cumsum().with_column('Chapter', np.arange(1, 44, 1))
cum_counts.plot(column_for_xticks=3)
plots.title('Cumulative Number of Times Each Name Appears', y=1.08);
```



在上图中，横轴显示章节号，纵轴显示每个字符在该章节被提及到的次数。

吉姆是核心人物，它的名字出现了很多次。请注意，第 30 章中汤姆出现并加入了哈克和吉姆，在此之前，汤姆在本书中几乎没有提及。他和吉姆的曲线在这个位置上迅速上升，因为涉及两者的行动都在变多。至于哈克，他的名字几乎没有出现，因为他是叙述者。

《小女人》是南北战争期间四个姐妹一起长大的故事。 在这本书中，章节号码拼写了出来，章节标题用大写字母表示。

```
# The chapters of Little Women, in a table

Table().with_column('Chapters', little_women_chapters)
```

Chapters
ONE PLAYING PILGRIMS "Christmas won't be Christmas witho ...
TWO A MERRY CHRISTMAS Jo was the first to wake in the gr ...
THREE THE LAURENCE BOY "Jo! Jo! Where are you?" cried Me ...
FOUR BURDENS "Oh, dear, how hard it does seem to take up ...
FIVE BEING NEIGHBORLY "What in the world are you going t ...
SIX BETH FINDS THE PALACE BEAUTIFUL The big house did pr ...
SEVEN AMY'S VALLEY OF HUMILIATION "That boy is a perfect ...
EIGHT JO MEETS APOLLYON "Girls, where are you going?" as ...
NINE MEG GOES TO VANITY FAIR "I do think it was the most ...
TEN THE P.C. AND P.O. As spring came on, a new set of am ...

（已省略 37 行）

我们可以跟踪主要人物的提及，来了解本书的情节。 主角乔（Jo）和她的姐妹梅格（Meg），贝丝（Beth）和艾米（Amy）经常互动，直到第 27 章中她独自搬到纽约。

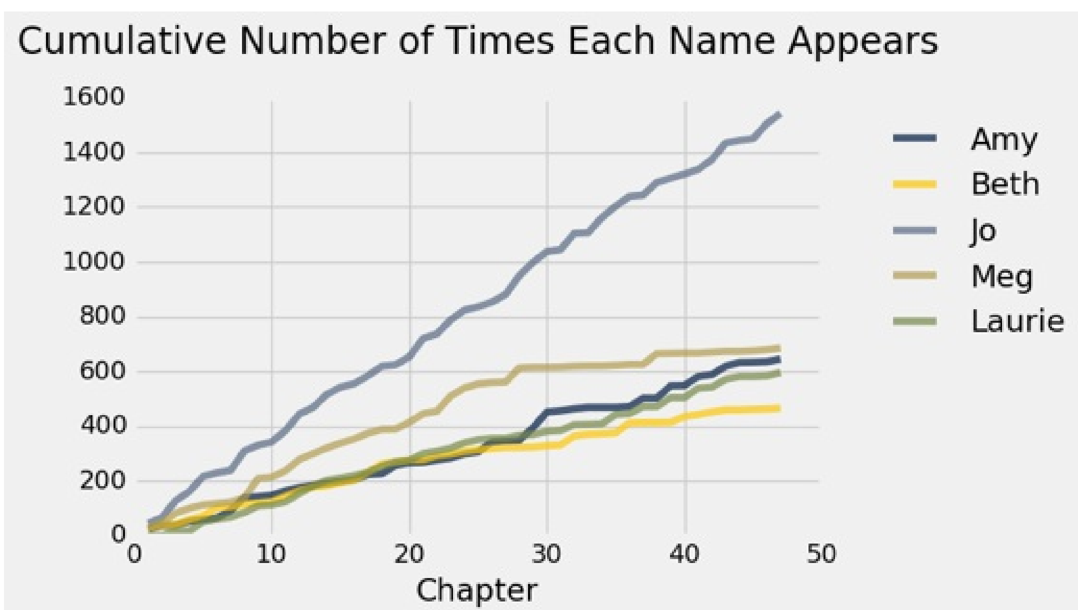
```
# Counts of names in the chapters of Little Women

counts = Table().with_columns([
        'Amy', np.char.count(little_women_chapters, 'Amy'),
        'Beth', np.char.count(little_women_chapters, 'Beth'),
        'Jo', np.char.count(little_women_chapters, 'Jo'),
        'Meg', np.char.count(little_women_chapters, 'Meg'),
        'Laurie', np.char.count(little_women_chapters, 'Laurie'),

    ])

# Plot the cumulative counts.

cum_counts = counts.cumsum().with_column('Chapter', np.arange(1, 48, 1))
cum_counts.plot(column_for_xticks=5)
plots.title('Cumulative Number of Times Each Name Appears', y=1.08);
```



劳里（Laurie）是个年轻人，最后和其中一个女孩结婚。 看看你是否可以使用这个图来猜测是哪一个。

另一种文本特征

在某些情况下，数量之间的关系能让我们做出预测。 本文将探讨如何基于不完整的信息做出准确的预测，并研究结合多种不确定信息来源进行决策的方法。

作为从多个来源获取信息的可视化的例子，让我们首先使用计算机来获取一些信息，它们通常手工难以获取。在小说的语境中，“特征”（Character）这个词有第二个含义：一个印刷符号，如字母，数字或标点符号。 在这里，我们要求计算机来计算《哈克贝利·芬》和《小女人》的每章中的字符和句号数量。

```
# In each chapter, count the number of all characters;
# call this the "length" of the chapter.
# Also count the number of periods.

chars_periods_huck_finn = Table().with_columns([
        'Huck Finn Chapter Length', [len(s) for s in huck_finn_chapters],
        'Number of Periods', np.char.count(huck_finn_chapters, '.')
    ])
chars_periods_little_women = Table().with_columns([
        'Little Women Chapter Length', [len(s) for s in little_women_chapters],
        'Number of Periods', np.char.count(little_women_chapters, '.')
    ])
```

这里是《哈克贝利·芬》的数据。 表格的每一行对应小说的一个章节，并显示章节中的字符和句号数量。 毫不奇怪，字符少的章节往往句号也少，一般来说 - 章节越短，句子越少，反之亦然。 然而，这种关系并不是完全可以预测的，因为句子的长度各不相同，还可能涉及其他标点符号，例如问号。

```
chars_periods_huck_finn
```

《哈克贝利·芬》章节长度	句号数量
7026	66
11982	117
8529	72
6799	84
8166	91
14550	125
13218	127
22208	249
8081	71
7036	70

（已省略 33 行）

这里是《小女人》的对应数据：

`chars_periods_little_women`

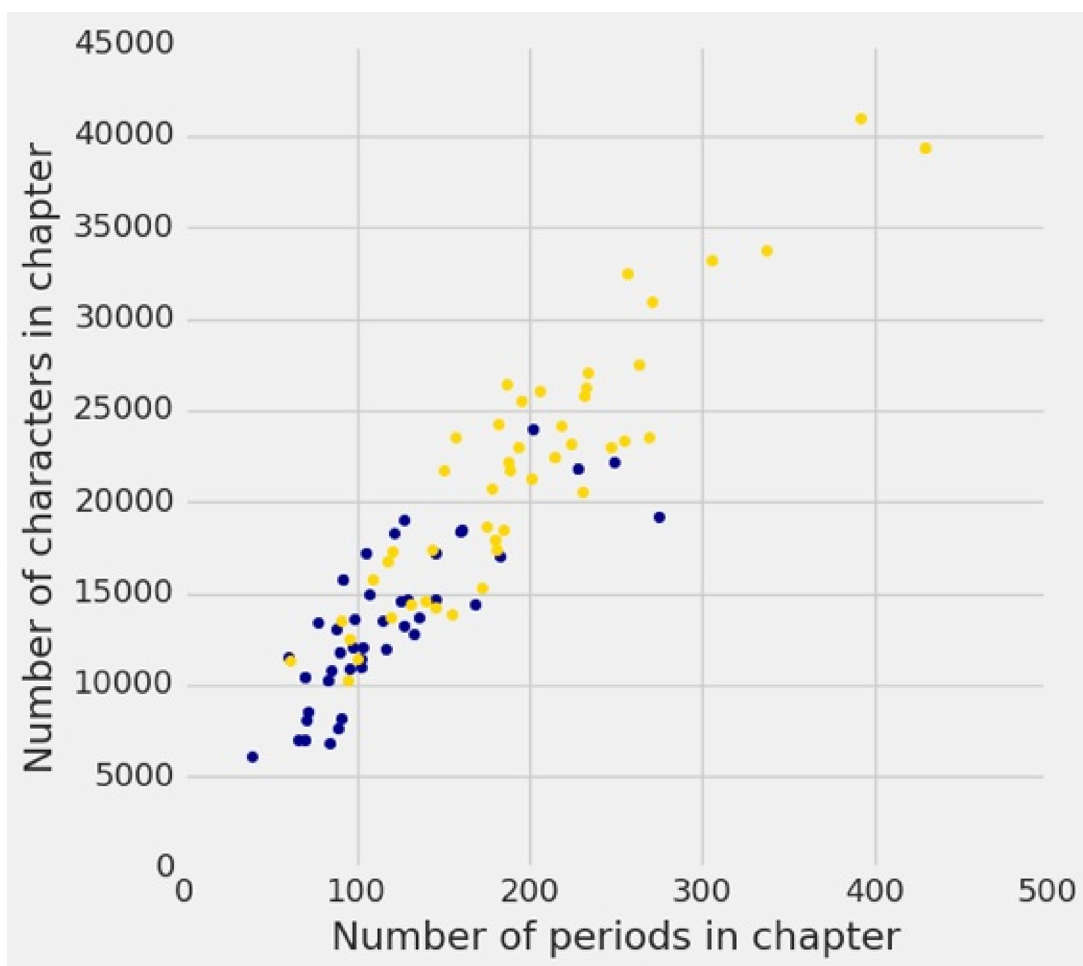
《小女人》章节长度	句号数量
21759	189
22148	188
20558	231
25526	195
23395	255
14622	140
14431	131
22476	214
33767	337
18508	185

（已省略 37 行）

你可以看到，《小女人》的章节总的来说比《哈克贝利·芬》的章节要长。让我们来看看这两个简单的变量 - 每一章的长度和句子数量 - 能否告诉我们这两本书的更多内容。 我们实现它的一个方法是在同一个图上绘制两组数据。

在下面的图中，每本书的每一章都有一个点。 蓝色圆点对应于《哈克贝利·芬》，金色圆点对应于《小女人》。横轴表示句号数量，纵轴表示字符数。

```
plots.figure(figsize=(6, 6))
plots.scatter(chars_periods_huck_finn.column(1),
              chars_periods_huck_finn.column(0),
              color='darkblue')
plots.scatter(chars_periods_little_women.column(1),
              chars_periods_little_women.column(0),
              color='gold')
plots.xlabel('Number of periods in chapter')
plots.ylabel('Number of characters in chapter');
```



这个绘图向我们展示，《小女人》的许多章节，而不是所有章节都比《哈克贝利·芬》的章节长，正如我们通过查看数字所看到的那样。 但它也向我们展示了更多东西。 请注意，蓝点粗略聚集在一条直线上，黄点也是如此。 此外看起来，两种颜色的点可能聚集在同一条直线上。

现在查看包含大约 100 个句号的所有章节。 绘图显示，这些章节大致包含约 10,000 个字符到约 15,000 个字符。每个句子大约有 100 到 150 个字符。

事实上，从这个绘图看来，这两本书的两个句号之间平均有 100 到 150 个字符，这是一个非常粗略的估计。 也许这两个伟大的 19 世纪小说正在表明我们现在非常熟悉的东西：Twitter 的 140 个字符的限制。

二、因果和实验

> 原文：Causality and Experiments
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

“这些问题已经，而且可能永远留在自然界难以捉摸的秘密之中，它们属于人类智力根本难以接近的一类问题。 - 1849 年 9 月，伦敦时报，霍乱如何传染和传播

死刑有威慑作用吗？巧克力对你有好处吗？什么导致乳腺癌？

所有这些问题试图为结果找到一个原因。仔细检查数据可以帮助揭示这些问题。在本节中，你将学习建立因果关系所涉及的一些基本概念。

观察是良好科学的关键。观察研究是一项研究，科学家根据他们所观察到的，但却无法产生的数据作出结论。在数据科学中，许多这样的研究涉及对一组个体的观察，称为实验的利害关系（factor of interest），以及对每个个体的测量结果。

将个体视为人是最容易的。在研究巧克力是否对健康有好处时，个体确实是人，实验是吃巧克力，结果可能是血压的测量。但观察研究中的个体不一定是人。在研究死刑是否具有威慑作用时，个体可以为联盟的 50 个州。允许死刑的州的法律是实验，结果可能是州的谋杀率。

根本问题是实验是否对结果有影响。实验和结果之间的任何关系被称为关联。如果实验导致结果发生，那么这个关联是因果关系。因果关系是本节开头提出的所有三个问题的核心。例如，问题之一是巧克力是否直接导致健康状况的改善，而不是巧克力与健康之间是否存在关联。

因果关系的建立往往分两个阶段进行。首先，观察一个关联。接下来，更仔细的分析决定了因果关系。

John Snow 和 Broad 街水泵

观察和可视化：John Snow 和 Broad 街水泵

精确观察导致建立因果关系的例子之一，最早可以追溯到 150 多年前。为了将你的思维带回正确的时间，试着想象一下 19 世纪 50 年代的伦敦。这是世界上最富裕的城市，但其中许多人却极度贫困。那时，查尔斯·狄更斯（Charles Dickens）在名气鼎盛时，正在写作关于他们的困境的文章。这个城市的贫困地区疾病盛行，霍乱是最可怕的。那个时候还不知道细菌会

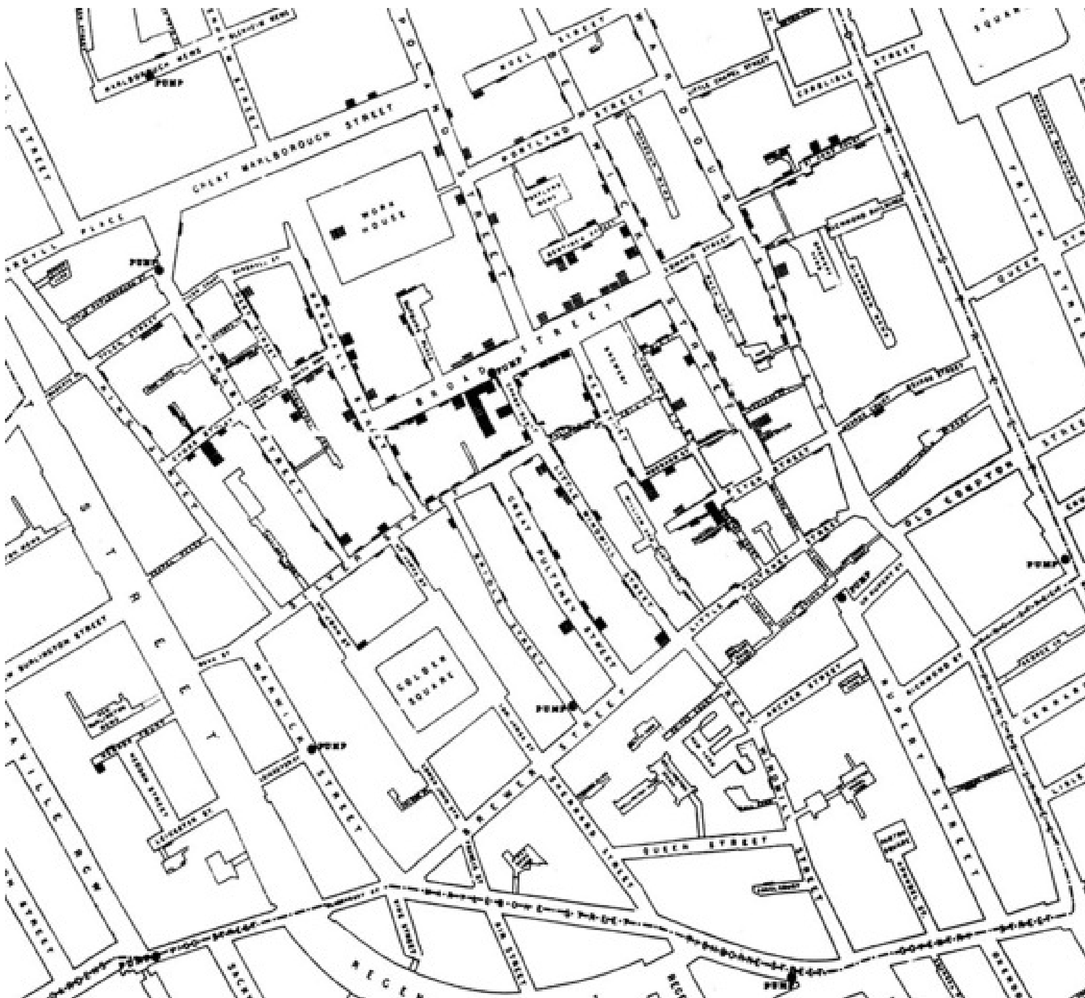
导致疾病，主流理论是“瘴气”是主要的罪魁祸首。 瘴气表现为恶臭，被认为是由腐烂物质引起的无形的有毒颗粒。伦敦的部分地区气味非常糟糕，特别是在炎热的天气里。为了保护自己免受感染，那些有能力的人把甜的东西放在鼻子上。

几年来，一个名叫约翰·斯诺（John Snow）的医生一直在跟踪着时不时袭击英国的巨大霍乱。疾病突然到来，几乎立即致命：人们在一两天内死亡，数百人在一个星期内死亡，单批总死亡人数可能达到数万人。斯诺对瘴气理论持怀疑态度。他注意到，当整个家庭被霍乱摧毁时，邻居有时完全没有受到影响。当他们呼吸和邻居一样的空气和瘴气时，不好的气味和霍乱的发生之间没有什么紧密的联系。

斯诺还注意到，这种疾病的发作几乎总是牵涉呕吐和腹泻。因此，他认为这种感染是由人们吃或喝的东西来进行的，而不是他们所呼吸的空气。他主要怀疑被污染的水。

1854 年 8 月底，霍乱在过度拥挤的伦敦苏豪区爆发。随着死亡人数的增加，斯诺用一种在疾病传播研究中成为标准的方法，勤奋地将它们记录下来：他画了一张地图。在该地区的街道地图上，他记录了每次死亡的地点。

这是斯诺的原始地图。每个黑色条形代表一次死亡。黑色圆圈标记了水泵的位置。地图上显示了一个惊人的启示 - 死亡大致集中在 Broad 街水泵周围。



斯诺仔细研究了他的地图，并调查了明显的异常。他们都设计 Broad 街水泵。例如：

- 死亡发生在离 Rupert 街水泵更近的房子，而不是 Broad 街。尽管 Rupert 街水泵直线上更近，但由于街道布局不方便，是死路一条。那些房子里的居民使用了 Broad 街水泵。
- 泵东边的两个街区没有死亡。那是 Lion Brewery 的位置，那里的工人喝了他们酿造的东西。如果他们想喝水，啤酒厂有自己的井。
- Broad 街水泵几个街区之外的房子里，发生了少量死亡。那些孩子在上学路上从 Broad 街水泵饮水。泵的水清凉爽口。

最后一个支持斯诺的理论的证据是，在距离 Soho 区很远的 Hampstead 地区的两个孤立的死亡事件。斯诺对这些人感到困惑，直到他得知死者是住在 Broad 街的 Susannah Eley 夫人和她的侄女。Eley 夫人每天都将 Broad 街水泵的水带到 Hampstead 给她。她喜欢水的味道。

后来发现了一个粪坑，距离 Broad 街水泵几英尺远，渗入了井里面。因此，来自霍乱受害者房子的污水污染了水泵的水。

斯诺用他的地图来说服当地政府，拆除 Broad 街水泵的手柄。虽然霍乱疫情已经在减少，但是停止使用这种水泵有可能阻止了许多人死于未来的疾病。

Broad 街水泵的手柄的拆除已成为一个传奇。在亚特兰大的疾病控制中心（CDC），当科学家寻找流行病问题的简单答案时，他们有时会互相问：“这个水泵的手柄在哪里？”

斯诺的地图是数据可视化的最早和最强大的用法之一。现在各种疾病地图是跟踪流行病的标准工具。

因果关系

虽然地图给了斯诺强有力的证据，说明了供水的清洁是控制霍乱的关键，但是，为了使“污染的水导致疾病的传播”这个科学论证有说服力，还有很长一段路要走。为了使案例更有说服力，他必须使用比较法。

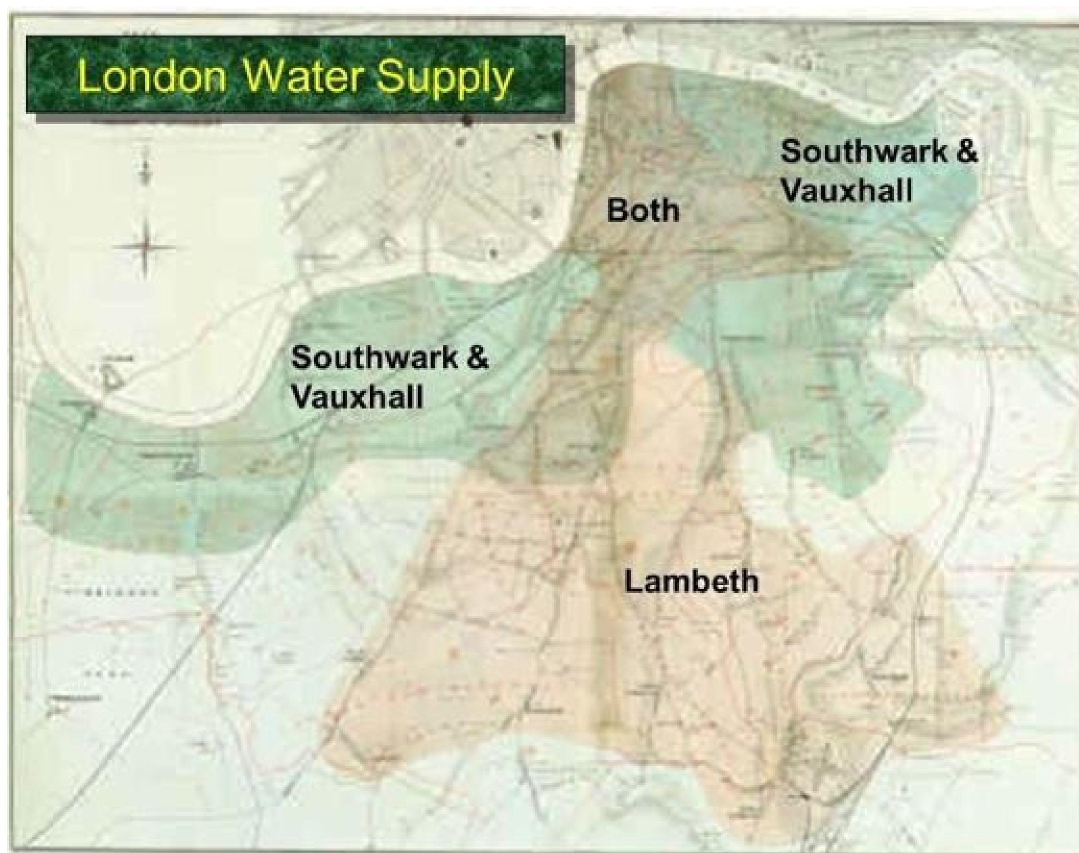
科学家使用比较来确定实验与结果之间的关联。他们比较了一组接受实验的个体（实验组）的结果，和一组没有接受实验的个体的结果（对照组）。例如，现在的研究人员可能会比较死刑国家和没有死刑的国家的平均谋杀率。

如果结果不同，那就是表明关联的证据。但是为了确定因果关系，需要更加小心。

斯诺的“大实验”

斯诺为自己在 Soho 中学到的东西感到鼓舞，他对霍乱的死亡情况做了更彻底的分析。一段时间中，他一直在收集伦敦一个地区的数据，这里由两家水厂服务。Lambeth 水厂从污水排入泰晤士河的地方的上游抽水。它的水比较干净。但 Southwark and Vauxhall (S&V) 公司在污水排放的下游抽水，因此其供水受到污染。

下图显示了两家公司所服务的地区。斯诺专注于两个服务地区重叠的地方。



斯诺注意到，S&V 供应的人和 Lambeth 供应的人之间没有系统的差别。“每家公司都供应富人和穷人，大房子和小房子，接受不同公司的供水的人的状况或职业并没有差别......接受两家公司供水的人或者房子都没什么区别，它们周围的物理状况也没什么区别...”

唯一的区别是供水方面，“一组供水含有伦敦的污水，其中有一些可能来自霍乱病人，另一组则不含。”

斯诺相信他能够得出一个清楚的结论，斯诺在下表中总结了他的数据。

Supply Area	Number of houses	cholera deaths	deaths per 10,000 houses
S&V	40,046	1,263	315
Lambeth	26,107	98	37
Rest of London	256,423	1,422	59

数字在指责 S&V。 S&V 供应的房屋的霍乱死亡率几乎是 Lambeth 供应的房屋的十倍。

建立因果

用本节前面提出的语言，可以将 S&V 房屋中的人作为实验组，Lambeth 房屋中的人作为对照组。斯诺的分析中的一个关键因素是，除了实验组以外，两组相互比较。

为了确定供水是否引起霍乱，斯诺必须比较两个彼此相似的群体，它们只有一方面不同：供水。只有这样，他才能够将其结果的差异归因于供水。如果这两个群体在其他方面有所不同，那么就很难把供水视为疾病的来源。例如，如果实验组由工厂工人组成，而对照组不是，那么两组之间的结果之间的差异可能是由于供水，工厂工作或两者兼有，或使两组彼此不同的其它因素。最后的图景会更加模糊。

斯诺的才智在于，确定可以使他的比较清晰的两组。他开始着手建立水污染和霍乱感染之间的因果关系，并且在很大程度上他成功了，尽管瘴气学说忽视甚至嘲笑他。当然，斯诺并不了解人类感染霍乱的详细机制。这个发现是在 1883 年，当时德国科学家罗伯特·科赫（Robert Koch）分离出霍乱弧菌，这种霍乱弧菌是进入人体小肠并引起霍乱的细菌。

事实上，霍乱弧菌在 1854 年由意大利的菲利波·帕齐尼（Filippo Pacini）发现，就在斯诺在伦敦分析他的数据的时候。由于意大利瘴气学说的统治，帕齐尼的发现并不为人所知。但到了十九世纪末，瘴气学说正在消失。随后的历史证明了帕齐尼和约翰·斯诺。斯诺的方法导致了流行病学领域的发展，它是疾病传播的研究。

混淆

现在让我们回到更现代化的时代，带着我们一路上学到的重要经验：

在一项观察研究中，如果实验组和对照组在实验以外的方面有所不同，则很难对因果关系作出结论。

两组之间的根本区别（除了实验）被称为混淆因素，因为当你试图得出结论时，它可能会混淆你（也就是搞砸你）。

示例：咖啡和肺癌。二十世纪六十年代的研究表明，喝咖啡的人患肺癌的比率高于不喝咖啡的人。因此，有些人认为咖啡是肺癌的一个原因。但咖啡不会导致肺癌。分析包含一个混淆因素 - 吸烟。在那些日子里，喝咖啡的人也可能是吸烟者，吸烟确实会导致肺癌。喝咖啡与肺癌有关，但不会导致疾病。

混淆因素在观察研究中很常见。良好的研究需要非常小心，以减少混淆。

随机化

避免混淆的一个很好的方法是，将个体随机分配到实验和对照组，然后将实验给予分配到实验组的人。随机化使两组除了实验之外都相似。

如果你能够将个体随机分为实验组和对照组，你正在进行一项随机对照试验（RCT）。有时候，人们在实验中的反应会受到他们知道他们在哪个群体的影响。所以你可能希望进行盲法实验，其中个体不知道他们是在实验组还是对照组。为了使它有效，你必须把安慰剂给控制组，这是一种和实验看起来完全一样的东西，但实际上没有效果。

随机对照实验早已成为医学领域的黄金标准，例如确定新药是否有效。在经济学等其他领域也越来越普遍。

示例：墨西哥的福利补贴。在 20 世纪 90 年代的墨西哥村庄，贫困家庭的孩子往往没有入学。其中一个原因是年龄较大的孩子可以上班，从而帮助家庭。墨西哥财政部长 Santiago Levy 着手调查福利项目是否可以用来提升入学率和改善健康状况。他在一组村庄进行了一项随机对照试验，随机选择其中的一些来接受一个名为 PROGRESA 的新福利项目。如果他们的孩子定期上学，并且家庭使用了预防性医疗保险，那么这个项目会把钱捐给贫困家庭。如果孩子上中学而不是小学，会给他们更多钱，来补偿孩子的工资损失，女孩上学比男孩给的更多。其余的村庄没有得到这个实验，并形成了对照组。由于随机化，没有销魂因素，可以确定 PROGRESA 增加了入学率。对于男孩，入学率从对照组的 73% 上升到 PROGRESA 组的 77%。对于女孩来说，增长幅度更大，从对照组的 67% 增加到 PROGRESA 组的近 75%。由于这个实验的成功，墨西哥政府以 OPORTUNIDADES 这个新名称支持这个项目，作为对一个健康和受过良好教育的人口的投资。

在某些情况下，即使目标是调查因果关系，也不可能进行随机对照实验。例如，假设你想研究怀孕期间饮酒的影响，你随机将一些孕妇分配到你的“酒精”组。如果你给他们喝一杯，你不应该期待她们会合作。在这种情况下，你几乎总是在进行观察研究，而不是实验。要警惕混淆因素。

尾注

根据我们开发的术语，约翰·斯诺进行了一项观察研究，而不是一个随机的实验。但是他把自己的研究称为“大实验”，因为他写道：“至少三十万人......被分成两组，他们无法选择，在大多数情况下，他们并不知情......”

斯诺的这种研究有时被称为“自然实验”。然而，真正的随机化并不仅仅意味着，实验和对照组“在他们无法选择的情况下”进行选择。

随机化的方法可以像掷硬币一样简单。它也可能更复杂一点。但是随机化的每一种方法都是由一系列精心定义的步骤组成的，这些步骤允许几率以数学方式指定。这有两个重要的结果。

- 它使我们能够以数学方式，计算随机化产生实验和对照组的可能性。
- 它使我们能够对实验组和对照组之间的差异作出精确的数学表述。这反过来帮助我们对实验是否有效作出正确的结论。

在本课程中，你将学习如何进行和分析你自己的随机实验。这将涉及比本节更多的细节。目前，只需关注主要思想：尝试建立因果关系，如果可能，进行随机对照实验。如果你正在进行一项观察研究，你可能能够建立联系而不是因果关系。在根据观察研究得出因果关系的结论之前，要非常小心混淆因素。

术语

- observational study：观察研究
- treatment：实验
- outcome：结果
- association：关联/联系
- causal association：因果联系
- causality：因果（关系）
- comparison：比较
- treatment group：实验组
- control group：对照组
- epidemiology：流行病学/传染病学
- confounding：混淆
- randomization：随机化
- randomized controlled experiment：随机对照实验
- randomized controlled trial (RCT)：随机对照实验
- blind：盲法
- placebo：安慰剂

有趣的事实

- 约翰·斯诺有时被称为流行病学之父，但他是专业的麻醉师。 他的病人之一是维多利亚女王，她是分娩时麻醉剂的早期接受者。

- 弗洛伦斯·南丁格尔，现代护理实践的创始人，因其在克里米亚战争中的工作而闻名，是一位顽固瘴气主义者。 她没有时间研究传染病和细菌的理论，也没有时间讲述她的话。 她说：“与这个学说相关的荒谬是无穷无尽的。一言以蔽之，从一般意义上说，没有任何科学研究可以接受的证据表明，存在传染病这样的事情。”

- 后来的随机对照试验表明，PROGRESA 坚持的条件 - 孩子上学，预防性医疗保险 - 对于提升入学率没有必要。 只是提高福利金就足够了。

扩展阅读

- The Strange Case of the Broad Street Pump: John Snow and the Mystery of Cholera 由 Sandra Hempel 所著，加利福尼亚大学出版社出版，读起来像是侦探小说。 这是本节中约翰·斯诺和他的工作的主要来源之一。 一些警告：这本书的一些内容令人反胃。

- Poor Economics 由 MIT 的 Abhijit V. Banerjee 和 Esther Duflo 所著的畅销书，是对抗全球贫困的方式的易理解的真实记录。 它包含了很多 RCT 的例子，包括本节中的 PROGRESA 示例。

三、Python 编程

> 原文：Programming in Python
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

编程可以极大地提高我们收集和分析世界信息的能力，而这些信息又可以通过上一节所述的谨慎推理来发现。 在数据科学中，编写程序的目的是，指示计算机执行分析步骤。 电脑无法自行研究世界。 人们必须准确描述计算机应该执行什么步骤来收集和分析数据，这些步骤是通过程序来表达的。

表达式

编程语言比人类语言简单得多。 尽管如此，在任何语言中，还是有一些语法规则需要学习，这里就是我们开始的地方。 在本文中，我们将使用 Python 编程语言。 学习语法规则是必不可少的，最基本的程序中使用的规则也是更复杂程序的核心。

程序由表达式组成，向计算机描述了如何组合数据片段。 例如，乘法表达式由两个数字表达式之间的 `*` 符号组成。表达式，例如 `3*4`，由计算机求值。在这种情况下，（IPython 中的）每个单元格中的最后一个表达式的值（求值结果）将显示在单元格下方，这里是 12。

```
3 * 4
12
```

编程语言的语法规则是僵化的。 在 Python 中， `*` 符号不能连续出现两次。 计算机不会试图解释一个与规定的表达式结构不同的表达式。 相反，它会显示 `SyntaxError` 错误。 语言的语法是其语法规则的集合， `SyntaxError` 表示表达式结构不匹配任何语法规则。

```
3 * * 4
  File "<ipython-input-4-d90564f70db7>", line 1
    3 * * 4
        ^
SyntaxError: invalid syntax
```

表达式的小改动可以完全改变它的含义。 下面， `*` 之间的空格已被删除。 因为 `**` 出现在两个数字表达式之间，所以表达式是一个格式良好的指数表达式（第一个数字的第二个数字次方， `3*3*3*3` ）。 符号 `*` 和 `**` 称为运算符，它们组合的值称为操作数。

```
3 ** 4
81
```

常用操作符。 数据科学通常涉及数值的组合，而编程语言中的一组操作符，是为了使得表达式可以用于表示任何类型的算术。 在Python中，以下操作符是必不可少的。

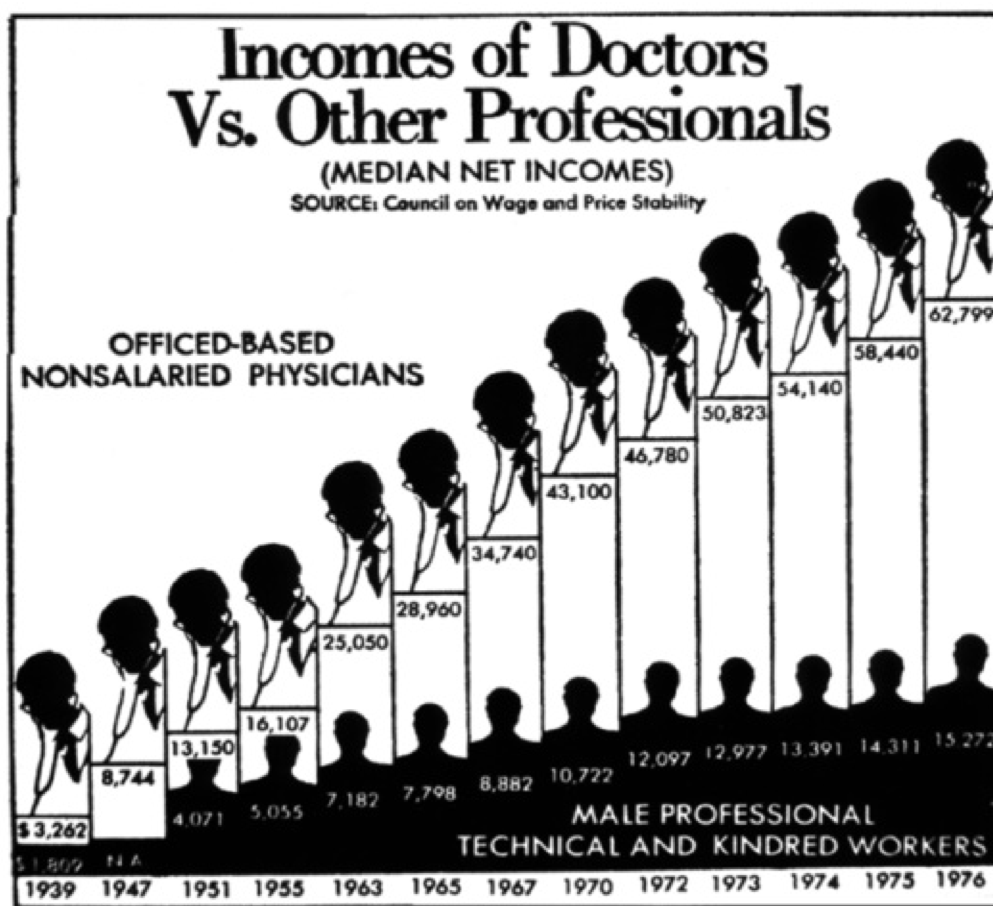
表达式类型	运算符	示例	値
加法	`+`	`2 + 3`	5
减法	`-`	`2 - 3`	-1
乘法	`*`	`2 * 3`	6
除法	`/`	`7 / 3`	2.66667
取余	`%`	`7 % 3`	1
指数	`**`	`2 ** 0.5`	1.41421

Python 表达式遵循熟悉的优先级规则，与代数中相同：乘法和除法在加法和减法之前计算。 圆括号可以用来在较大的表达式中，将较小的表达式组合在一起。

```
1 + 2 * 3 * 4 * 5 / 6 ** 3 + 7 + 8 - 9 + 10
17.555555555555557
1 + 2 * (3 * 4 * 5 / 6) ** 3 + 7 + 8 - 9 + 10
2017.0
```

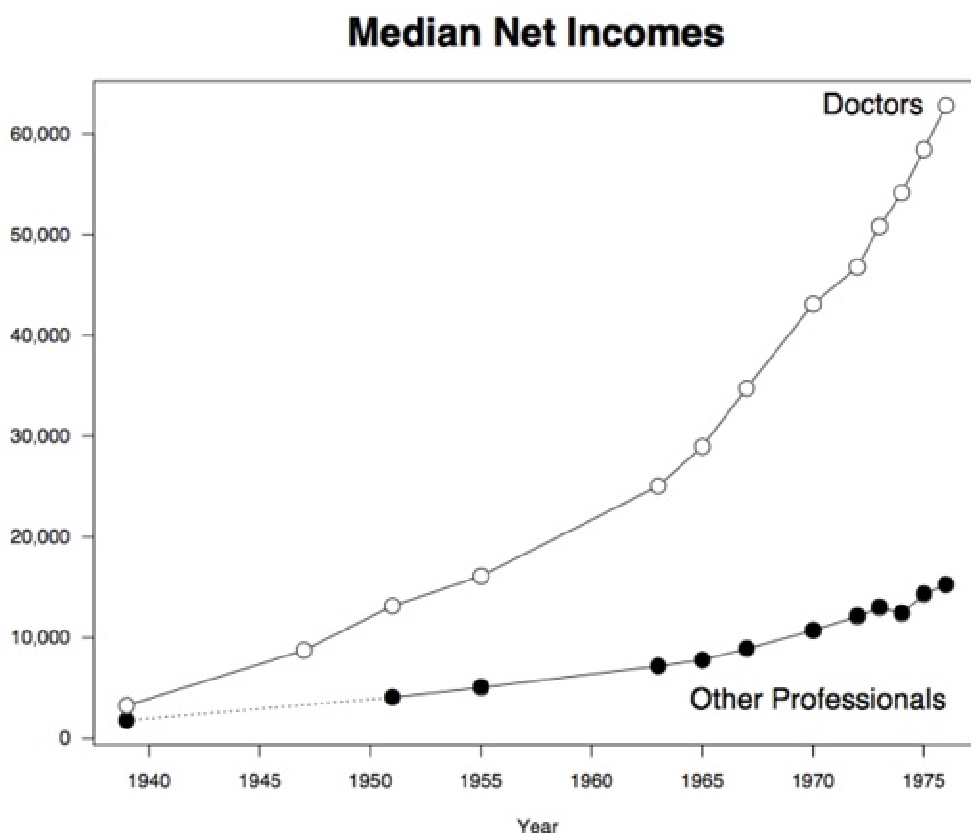
示例

这里是一个图表，来自 20 世纪 80 年代初期的“华盛顿邮报”（The Washington Post），试图比较几十年来医生的收入与其他专业人员的收入。 我们是否真的需要在每个条形上看到两个头（一个带有听诊器）？ 耶鲁大学教授爱德华·图夫特（Edward Tufte）是世界上量化信息可视化的专家之一，他为这种不必要的修饰创造了“垃圾图表”（chartjunk）一词。 这张图也是 Tufte 痛恨的“数据与油墨比例过低”的一个例子。



华盛顿邮报图片

最重要的是，图的横轴不是按比例绘制的。 这对条形图的形状有显着的影响。 当按规模绘制并把装饰修剪掉时，图表显示的趋势非常不同于原来明显的线性增长。 下面的优雅图表由统计系统 R 的创始人之一 Ross Ihaka 提供。



Ross Ihaka 的图片版本

在 1939 年到 1963 年间，医生的收入从 3,262 美元增加到 25,050 美元。 所以在这个时期，每年的平均收入增加了大约 900 美元。

```
(25050 - 3262)/(1963 - 1939)
907.8333333333334
```

在 Ross Ihaka 的图表中可以看到，在这个时期，医生的收入大致呈线性上升，并且保持在一个相对稳定的水平。 正如我们刚刚计算的那样，这个比率大约是 900 美元。

但是从 1963 年到 1976 年，这个比例是三倍多：

```
(62799 - 25050)/(1976 - 1963)
2903.769230769231
```

这就是 1963 年之后，这个图形急剧上升的原因。

本章介绍了许多类型的表达式。 学习编程需要结合学到所有的东西，调查计算机的行为。 如果你连续除两次会发生什么？ 你并不需要总是问专家（或互联网）；许多这些细节可以通过自己尝试发现。

数值

整数值

计算机为执行数值计算而设计，但是关于处理数字有一些重要的细节，每个处理定量数据的程序员都应该知道它。 Python（和大多数其他编程语言）区分两种不同类型的数字：

- 整数在 Python 语言中称为 `int` 值。 它们只能表示没有小数部分的整数（负数，零或正数）
- 实数在 Python 语言中被称为 `float` 值（或浮点值）。 他们可以表示全部或部分数字，但有一些限制。

数值的类型在展示方式上是明显的： `int` 值没有小数点， `float` 值总是有一个小数点。

```
# Some int values
2
2
1 + 3
4
-1234567890000000000
-1234567890000000000
# Some float values
1.2
1.2
1.5 + 2
3.5
3 / 1
3.0
-12345678900000000000.0
-1.23456789e+19
```

当一个 `float` 值和一个 `int` 值，通过算术运算符组合在一起时，结果总是一个 `float` 值。 在大多数情况下，两个整数的组合形成另一个整数，但任何数字（ `int` 或 `float` ）除以另一个将是一个 `float` 值。 非常大或非常小的 `float` 值可以使用科学记数法表示。

浮点值

浮点值非常灵活，但他们有限制。

`float` 可以表示非常大和非常小的数字。存在限制，但你很少遇到他们。 浮点数只能表示任何数字的 15 或 16 位有效数字；剩下的精度就会丢失。 这个有限的精度对于绝大多数应用来说已经足够了。 将浮点值与算术运算结合后，最后的几位数字可能不正确。 第一次遇到时，微小的舍入错误往往令人困惑。

第一个限制可以通过两种方式来观察。 如果一个计算的结果是一个非常大的数字，那么它被表示为无限大。 如果结果是非常小的数字，则表示为零。

```
2e306 * 10
2e+307
2e306 * 100
inf
2e-322 / 10
2e-323
2e-322 / 100
0.0
```

第二个限制可以通过涉及超过 15 位有效数字的表达式来观察。 在进行任何算术运算之前，这些额外的数字被丢弃。

```
0.6666666666666666 - 0.6666666666666666123456789
0.0
```

当两个表达式应该相等时，可以观察到第三个限制。 例如，表达式`2 ** 0.5`计算 2 的平方根，但是该值的平方不会完全恢复成 2。

```
2 ** 0.5
1.4142135623730951
(2 ** 0.5) * (2 ** 0.5)
2.0000000000000004
(2 ** 0.5) * (2 ** 0.5) - 2
4.440892098500626e-16
```

上面的最终结果是`0.0000000000000004440892098500626`，这个数字非常接近零。 这个算术表达式的正确答案是 0，但是最后的有效数字中的一个小错误，在科学记数法中显得非常不同。 这种行为几乎出现在所有的编程语言中，因为它是在计算机上进行算术运算的标准方式的结果。

尽管`float`并不总是精确的，但它们当然是可靠的，并且在所有不同种类的计算机和编程语言中，以相同的方式工作。

名称

名称通过赋值语句在 Python 中得到一个值。 在赋值中，名称后面是`=`，再后面是任何表达式。 `=`右边的表达式的值被赋给名称。 一旦名称有了赋给它的值，在将来的表达式中，值会替换为这个名称。

```
a = 10
b = 20
a + b
30
```

之前赋值的名称可以在`=`右边的表达式中使用。

```
quarter = 1/4
half = 2 * quarter
half
0.5
```

但是，仅仅是表达式的当前值赋给了名称。 如果该值稍后改变，则由该值定义的名称将不会自动更改。

```
quarter = 4
half
0.5
```

名称必须以字母开头，但可以包含字母和数字。 名称不能包含空格；相反，通常使用下划线字符 `_` 来替换每个空格。名称只在你编写的时候是有用的；程序员可以选择易于理解的名称。 通常，比起 `a` 和 `b` ，你可以创造更有意义的名字。 例如，为了描述加利福尼亚州伯克利 5 美元商品的销售税，以下名称阐明了各种相关数量的含义。

```
purchase_price = 5
state_tax_rate = 0.075
county_tax_rate = 0.02
city_tax_rate = 0
sales_tax_rate = state_tax_rate + county_tax_rate + city_tax_rate
sales_tax = purchase_price * sales_tax_rate
sales_tax
0.475
```

示例：增长率

相同数量在不同时间取得的两次测量值之间的关系通常表示为增长率。 例如，美国联邦政府在 2002 年雇用了 276.6 万人，在 2012 年雇用了 281.4 万人。为了计算增长率，我们必须首先决定将哪个值作为初始值。 对于随着时间变化的数值，较早的值是一个自然的选择。 然后，我们将变动值和初始值之间的差除以初始值。

```
initial = 2766000
changed = 2814000
(changed - initial) / initial
0.01735357917570499
```

通常从两个测量值的比例中减去 1，这产生相同的值。

```
(changed/initial) - 1
0.017353579175704903
```

这个值是 10 年间的增长率。 增长率的一个实用属性是，即使值以不同的单位表示，它们也不会改变。 所以，例如，我们可以以千人为单位，在 2002 年和 2012 年之间表达同样的关系。

```
initial = 2766
changed = 2814
(changed/initial) - 1
0.017353579175704903
```

10 年以来，美国联邦政府的雇员人数仅增长了 1.74%。 那个时候，美国联邦政府的总支出从 2.37 万亿美元增加到 2012 年的 3.38 万亿美元。

```
initial = 2.37
changed = 3.38
(changed/initial) - 1
0.4261603375527425
```

联邦预算增长 42.6% 远高于联邦雇员增长 1.74%。 实际上，联邦雇员的数量增长速度远远低于美国人口。美国人口同期增长 9.21%，从 2002 年的 2.8760 亿人增加到 2012 年的 3.41 亿。

```
initial = 287.6
changed = 314.1
(changed/initial) - 1
0.09214186369958277
```

增长率可能是负值，表示某种值的下降。 例如，美国的制造业就业岗位从 2002 年 的 1530 万减少到 2012 年的 1190 万，增长率为 -22.2%。

```
initial = 15.3
changed = 11.9
(changed/initial) - 1
-0.2222222222222222
```

年增长率是一年之内的某个数量的增长率。 年增长率为 0.035，累计十年，十年增长率为 0.41（即 41%）。

```
1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 - 1
0.410598760621121
```

相同的计算可以使用名称和指数表达。

```
annual_growth_rate = 0.035
ten_year_growth_rate = (1 + annual_growth_rate) ** 10 - 1
ten_year_growth_rate
0.410598760621121
```

同样，十年的增长率可以用来计算等价的年增长率。 下面，`t` 是两次测量值之间经过的年数。 下面计算过去 10 年联邦支出的年增长率。

```
initial = 2.37
changed = 3.38
t = 10
(changed/initial) ** (1/t) - 1
0.03613617208346853
```

十年来的总增长率相当于每年增长 3.6%。

总之，增长率 `g` 用来描述 `initial`（初始值）和经过一段时间 `t` 之后的 `changed`（变化值）的相对大小。 为了计算 `changed`，使用指数来重复应用增长率 `g` `t` 次。

```
initial * (1 + g) ** t
```

为了计算 `g`，计算总增长率的 `1/t` 次方并减一。

```
(changed/initial) ** (1/t) - 1
```

调用表达式

调用表达式调用函数，这些函数是具名操作。 函数名称首先出现，然后是括号中的表达式。

```
abs(-12)
12
round(5 - 1.3)
4
max(2, 2 + 3, 4)
5
```

在这最后一个例子中，`max` 函数在三个参数：`2`，`5` 和 `4` 上调用。圆括号内每个表达式的值被传递给函数，函数返回整个调用表达式的最终值。 `max` 函数可以接受任意数量的参数并返回最大值。

一些函数默认是可用的，比如 `abs` 和 `round`，但是大部分内置于 Python 语言的函数都存储在一个称为模块的函数集合中。 导入语句用于访问模块，如 `math` 或 `operator`。

```
import math
import operator
math.sqrt(operator.add(4, 5))
3.0
```

可以使用 `+` 和 `**` 运算符来表达等价的表达式。

```
(4 + 5) ** 0.5
3.0
```

运算符和调用表达式可以在表达式中一起使用。两个值之间的百分比差异用于比较一些值，它们明显既不是 `initial` 也不是 `changed`。例如，2014 年，佛罗里达农场生产了 27.2 亿个蛋，而爱荷华州农场生产了 162.5 亿个鸡蛋 [1]。百分比差值是数值之差的绝对值的 100 倍，再除以它们的平均值。在这种情况下，差值大于平均值，所以百分比差异大于 100。

> [1] http://quickstats.nass.usda.gov/

```
florida = 2.72
iowa = 16.25
100*abs(florida-iowa)/((florida+iowa)/2)
142.6462836056932
```

学习不同函数的行为，是学习编程语言的重要组成部分。Jupyter 笔记本可以帮助你记住不同函数的名称和效果。编辑代码单元格时，在输入名称的开头之后按 Tab 键，来显示补全该名称的方式列表。例如，在 `math` 后按 Tab 键，来查看 `math` 模块中所有可用函数。打字将缩小选项列表的范围。为了了解函数的更多信息，请在它的名称之后放置一个 `?`。例如，输入 `math.log` 将显示 `math` 模块中 `log` 函数的描述。

```
math.log?
log(x[, base])

Return the logarithm of x to the given base.
If the base not specified, returns the natural logarithm (base e) of x.
```

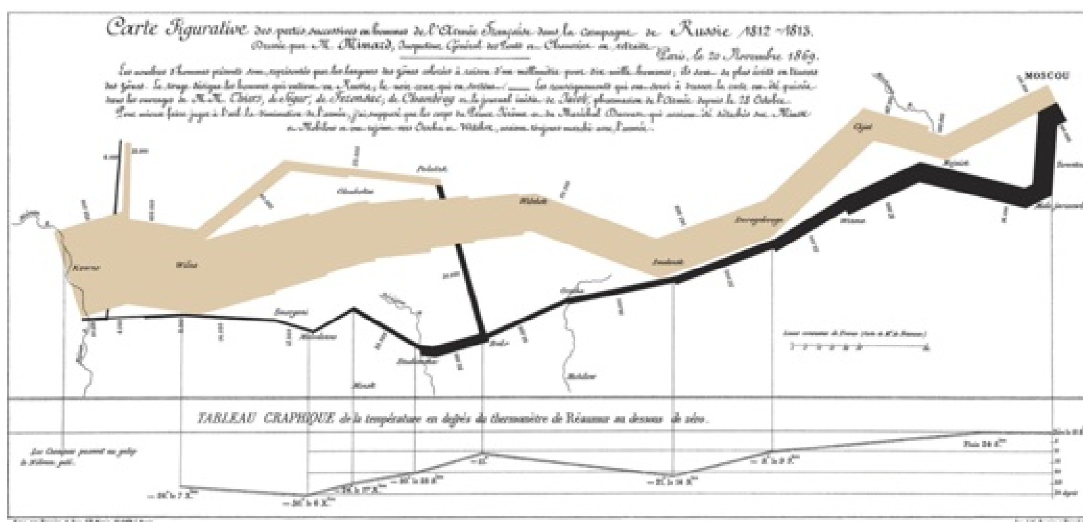
示例调用中的方括号表示参数是可选的。也就是说，可以用一个或两个参数来调用 `log`。

```
math.log(16, 2)
4.0
math.log(16)/math.log(2)
4.0
```

Python 的内建函数列表非常长，包含了许多在数据科学应用中不需要的函数。`math` 模块中的数学函数列表同样很长。本文将在上下文中介绍最重要的函数，而不是期望读者记住或理解这些列表。

示例

1869 年，一位名叫查尔斯·约瑟夫·米纳德（Charles Joseph Minard）的法国土木工程师，创造了一个图表，仍被认为是有史以来最伟大的图表之一。它显示了拿破仑军队从莫斯科撤退期间的损失。1812 年，拿破仑开始征服俄罗斯，他的军队中有超过 35 万人。他们确实到达了莫斯科，但是沿路一直受到损失的困扰。俄国军队不断撤退到俄罗斯深处，故意焚烧田野，并在撤退时摧毁村庄。这使法国军队在俄罗斯冬季来临之时，没有食物或避难所。法国军队在莫斯科没有取得决定性的胜利就撤退了。之后天气变冷，死了更多的人。回来的人还不到一万。



Minard 的地图

这个图表绘制在东欧地图上。 它始于左端的波兰-俄罗斯边界。 浅棕色的条形表示拿破仑的军队正在向莫斯科进军，黑色的条形代表军队的撤退。 在图表的每个点上，军队的宽度与军队中士兵的数量成正比。在图表的底部，Minard 包括了回程的温度。

注意当军队撤退时，黑色条形变窄。 渡过贝尔齐纳河是个特别的灾难，你能在图表上看到吗？

由于其简单和有力，这个图标是出色的。 Minard 展示了六个变量：

- 士兵的数量
- 行军的方向
- 位置的经纬度
- 回程的温度
- 十一月和十二月的具体日期的位置

Tufte 说 Minard 的图是“可能是有史以来最好的统计图表”。

这里是 Minard 数据的一个子集，取自 Leland Wilkinson 的 The Grammar of Graphics。

Longitude	Latitude	City	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

Minard 的子集

每一行表示特定位置的军队状态。 列以度为单位展示经度和纬度，位置的名称，军队是前进还是撤退，以及估计的人数。

在这个表格中，连续两个地点之间的人数的最大变化是在莫斯科撤退的时候，也是最大的百分比变化。

```
moscou = 100000
wixma = 55000
wixma - moscou
-45000
(wixma - moscou)/moscou
-0.45
```

在莫斯科的战斗中，人数下降了 45%。 换句话说，进入莫斯科的拿破仑的军队中，有几乎一半的人没有继续前进。

正如你在图表中看到的，Moiodexno 非常接近军队出发位置 Kowno。 在前进期间进入 Smolensk 的人中，只有不到 10% 的人在返回的途中到达了 Moiodexno。

```
smolensk_A = 145000
moiodexno = 12000
(moiodexno - smolensk_A)/smolensk_A
-0.9172413793103448
```

是的，只要使用没有名称的数字就可以做这些计算。 但是这些名称使得阅读代码和解释结果变得更容易。

值得注意的是，更大的绝对变化并不总是对应更大的百分比变化。

在前进期间，从 Smolensk 到 Dorogobouge 的绝对损失是 5000 人，而撤退期间，从 Smolensk 到 Orscha 的相应损失是 4000 人。

然而，Smolensk 和 Orscha 之间的百分比变化要大得多，因为，在撤退期间，Smolensk 的人员总数要小得多。

```
dorogobouge = 140000
smolensk_R = 24000
orscha = 20000
abs(dorogobouge - smolensk_A)
5000
abs(dorogobouge - smolensk_A)/smolensk_A
0.034482758620689655
abs(orscha - smolensk_R)
4000
abs(orscha - smolensk_R)/smolensk_R
0.16666666666666666
```

四、数据类型

> 原文：Data Types
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

每个值都有一个类型，内建的`type`函数返回任何表达式的结果的类型：

```
type(3)
int
type(3/1)
float
```

表达式的`type`是其最终值的类型。 所以，`type`函数永远不会表明，表达式的类型是一个名称，因为名称总是求值为它们被赋予的值。

```
x = 3
type(x) # The type of x is an int, not a name
int
```

我们已经遇到的另一种类型是内置函数。 Python 表明这个类型是一个`builtin_function_or_method`；函数和方法之间的区别在这个阶段并不重要。

```
type(abs)
builtin_function_or_method
```

这一章会探索其他实用的数据类型。

字符串

世界上大部分的数据都是文本，计算机中表示的文本被称为字符串。 字符串可以代表一个单词，一个句子，甚至是图书馆中每本书的内容。 由于文本可以包含数字（如`5`）或布尔值（`True`），字符串也可以描述这些东西。

表达式的含义取决于其结构和正在组合的值的类型。 因此，例如，将两个字符串加在一起会产生另一个字符串。 这个表达式仍然是一个加法表达式，但是它组合了一个不同类型的值。

```
"data" + "science"
'datascience'
```

加法完全是字面的；它将这两个字符串组合在一起而不考虑其内容。 它不增加空间，因为这些是不同的词；它取决于程序员（你）来指定。

```
"data" + " " + "science"
'data science'
```

单引号和双引号都可以用来创建字符串：`'hi'` 和 `"hi"` 是相同的表达式。 双引号通常是首选，因为它们允许在字符串中包含单引号。

```
"This won't work with a single-quoted string!"
"This won't work with a single-quoted string!"
```

为什么不能？ 试试看。

`str` 函数返回任何值的字符串表示形式。 使用此函数，可以构建具有嵌入值的字符串。

```
"That's " + str(1 + 1) + ' ' + str(True)
"That's 2 True"
```

字符串方法

可以使用字符串方法，从现有的字符串中构造相关的字符串，这些方法是操作字符串的函数。 这些方法通过在字符串后面放置一个点，然后调用该函数来调用。

例如，以下方法生成一个字符串的大写版本。

```
"loud".upper()
'LOUD'
```

也许最重要的方法是 `replace` ，它替换字符串中的所有子字符串的实例。 `replace` 方法有两个参数，即被替换的文本和替代值。

```
'hitchhiker'.replace('hi', 'ma')
'matchmaker'
```

字符串方法也可以使用变量名称进行调用，只要这些名称绑定到字符串。 因此，例如，通过首先创建 `"ingrain"` 然后进行第二次替换，以下两个步骤的过程从 `"train"` 生成 `"degrade"` 一词。

```
s = "train"
t = s.replace('t', 'ing')
u = t.replace('in', 'de')
u
'degrade'
```

注意 `t = s.replace('t', 'ing')` 的一行，不改变字符串 `s`，它仍然是 `"train"`。 方法调用 `s.replace('t', 'ing')` 只有一个值，即字符串 `"ingrain"`。

```
s
'train'
```

这是我们第一次看到方法，但是方法并不是字符串仅有的。 我们将很快看到，其他类型的对象可以拥有它们。

比较

布尔值通常来自比较运算符。 Python 包含了各种比较值的运算符。 例如，`3 > 1 + 1`。

```
3 > 1 + 1
True
```

值 `True` 表明这个比较是有效的；Python 已经证实了 `3` 和 `1 + 1` 之间关系的这个简单的事实。下面列出了一整套通用的比较运算符。

比较	运算符	True 示例	False 示例
小于	`<`	`2 < 3`	`2 < 2`
大于	`>`	`3 > 2`	`3 > 3`
小于等于	`<=`	`2 <= 2`	`3 <= 2`
大于等于	`>=`	`3 >= 3`	`2 >= 3`
等于	`==`	`3 == 3`	`3 == 2`
不等于	`!=`	`3 != 2`	`2 != 2`

一个表达式可以包含多个比较，并且为了使整个表达式为真，它们都必须有效。 例如，我们可以用下面的表达式表示 `1 + 1` 在 `1` 和 `3` 之间。

```
1 < 1 + 1 < 3
True
```

两个数字的平均值总是在较小的数字和较大的数字之间。 我们用下面的数字 `x` 和 `y` 来表示这种关系。 你可以尝试不同的 `x` 和 `y` 值来确认这种关系。

```
x = 12
y = 5
min(x, y) <= (x+y)/2 <= max(x, y)
True
```

字符串也可以比较，他们的顺序是字典序。 较短的字符串小于以较短的字符串开头的较长的字符串。

```
"Dog" > "Catastrophe" > "Cat"
True
```

序列

值可以分组到集合中，这允许程序员组织这些值，并使用单个名称引用它们中的所有值。 通过将值分组在一起，我们可以编写代码，一次执行许多数据计算。

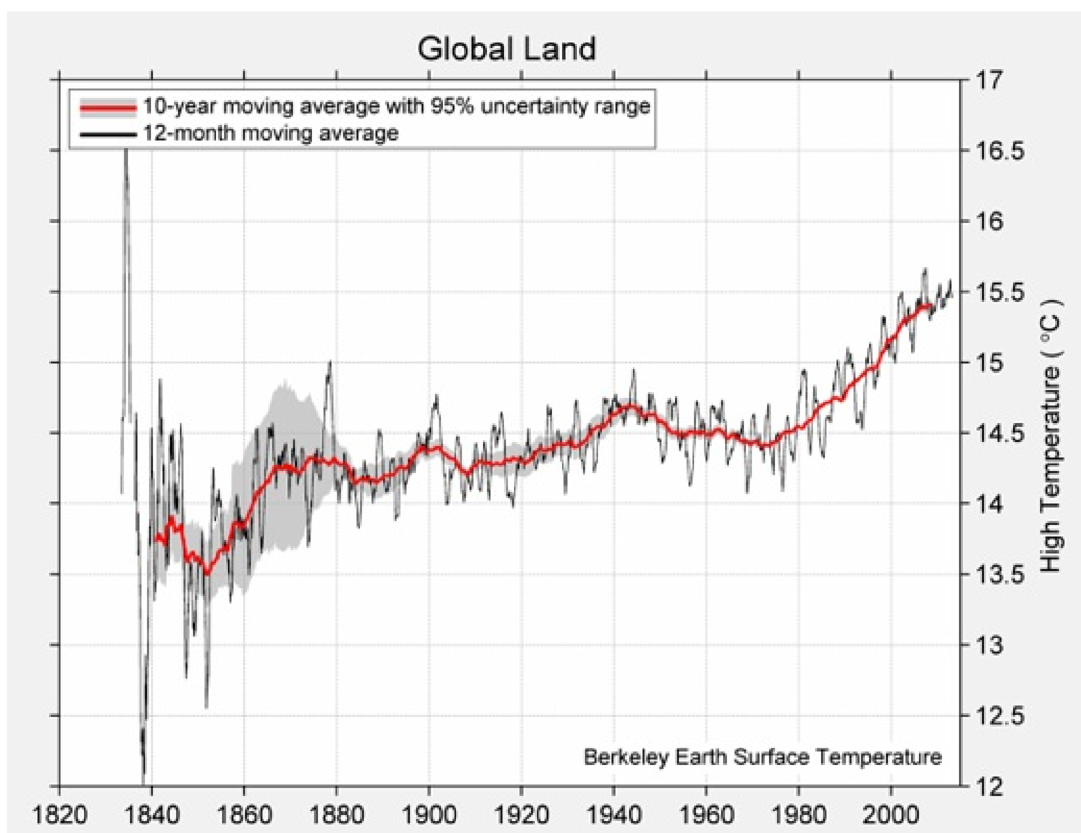
在几个值上调用 `make_array` 函数，将它们放到一个数组中，这是一种顺序集合。 下面，我们将四个不同的温度收集到一个名为 `temps` 的数组中。 这些分别是 1850 年，1900 年，1950 年和 2000 年的几十年间，地球上所有陆地的估计日平均绝对高温（摄氏度），表示为 1951 年至 1980 年间平均绝对高温的偏差，为 14.48 度。

集合允许我们使用单个名称，将多个值传递给一个函数。 例如， `sum` 函数计算集合中所有值的和， `len` 函数计算其长度。 （这是我们放入的值的数量。）一起使用它们，我们可以计算一个集合的平均值。

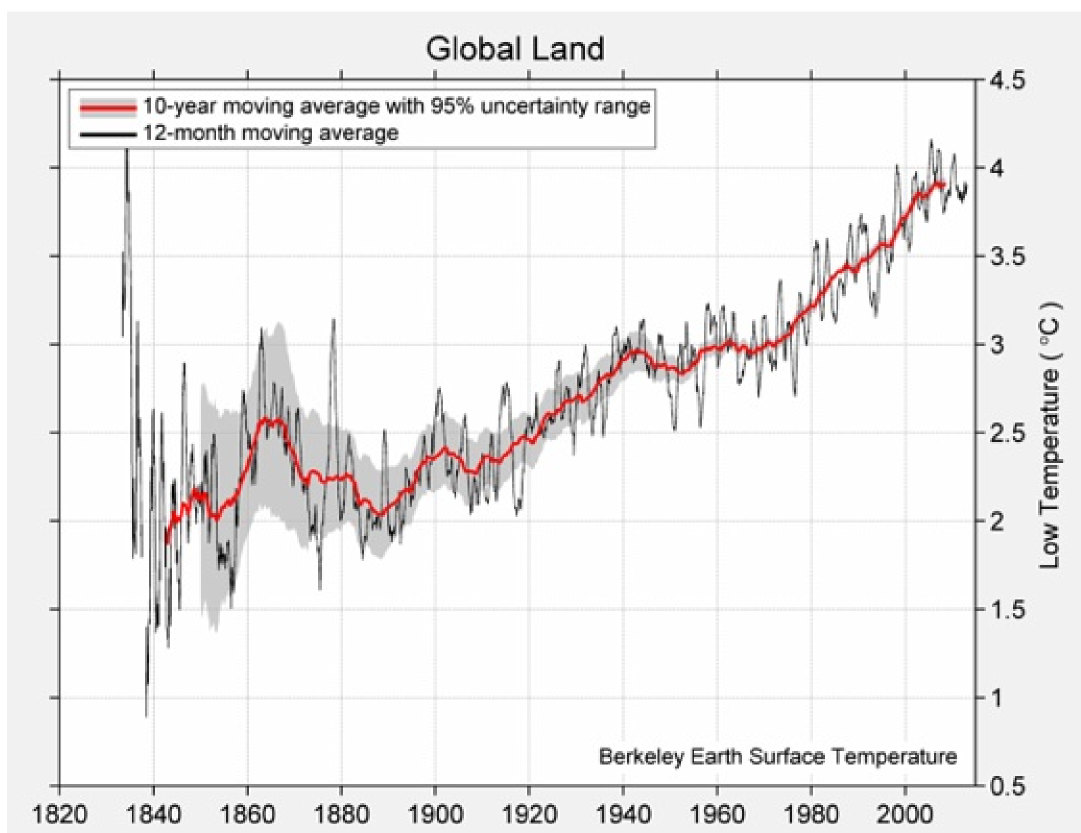
```
sum(highs)/len(highs)
14.434000000000001
```

日高温和低温的完整图表在下面。

日高温均值


Global Land
10-year moving average with 95% uncertainty range
12-month moving average
Berkeley Earth Surface Temperature
High Temperature (°C)
17
16.5
16
15.5
15
14.5
14
13.5
13
12.5
12
1820
1840
1860
1880
1900
1920
1940
1960
1980
2000

日低温均值


Global Land
10-year moving average with 95% uncertainty range
12-month moving average
Berkeley Earth Surface Temperature
Low Temperature (°C)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
1820
1840
1860
1880
1900
1920
1940
1960
1980
2000

数组

Python 中有很多种类的集合，我们在这门课中主要使用数组。 我们已经看到， `make_array` 函数可以用来创建数值的数组。

数组也可以包含字符串或其他类型的值，但是单个数组只能包含单一类型的数据。 （无论如何，把不相似的数据组合在一起，通常都没有意义）。例如：

```
english_parts_of_speech = make_array("noun", "pronoun", "verb", "adverb", "adjective",
"conjunction", "preposition", "interjection")
english_parts_of_speech
array(['noun', 'pronoun', 'verb', 'adverb', 'adjective', 'conjunction',
       'preposition', 'interjection'],
      dtype='<U12')
```

> 译者注：
>
> ```
> import numpy as np
> make_array = lambda *args: np.asarray(args)
> ```

返回到温度数据，我们创建 1850 年，1900 年，1950 年和 2000 年的几十年间，日平均高温的数组。

```
baseline_high = 14.48
highs = make_array(baseline_high - 0.880,
                   baseline_high - 0.093,
                   baseline_high + 0.105,
                   baseline_high + 0.684)
highs
array([ 13.6  ,  14.387,  14.585,  15.164])
```

数组可以用在算术表达式中来计算其内容。 当数组与单个数组合时，该数与数组的每个元素组合。 因此，我们可以通过编写熟悉的转换公式，将所有这些温度转换成华氏温度。

```
(9/5) * highs + 32
array([ 56.48  ,  57.8966,  58.253 ,  59.2952])
```

highs

(9/5) * [13.6, 14.387, 14.585, 15.164] + 32 = [(9/5) * 13.6 + 32, (9/5) * 14.387 + 32, (9/5) * 14.585 + 32, (9/5) * 15.164 + 32] = [56.48, 57.8966, 58.253, 59.2952]

数组也有方法，这些方法是操作数组值的函数。 数值集合的均值是其总和除以长度。 以下示例中的每对括号都是调用表达式的一部分；它调用一个无参函数来对数组 `highs` 进行计算。

```
highs.size
4
highs.sum()
57.736000000000004
highs.mean()
14.434000000000001
```

数组上的函数

`numpy` 包，在程序中缩写为 `np`，为 Python 程序员提供了创建和操作数组的，方便而强大的函数。

```
import numpy as np
```

例如，`diff` 函数计算数组中每两个相邻元素之间的差。 差数组的第一个元素是原数组的第二个元素减去第一个元素。

```
np.diff(highs)
array([ 0.787,  0.198,  0.579])
```

完整的 Numpy 参考详细列出了这些功能，但一个小的子集通常用于数据处理应用。 它们分组到了 `np` 中不同的包中。 学习这些词汇是学习 Python 语言的重要组成部分，因此在你处理示例和问题时，请经常回顾这个列表。

但是，你不需要记住这些，只需要将它用作参考。

每个这些函数接受数组作为参数，并返回单个值。

函数	描述
`np.prod`	将所有元素相乘
`np.sum`	将所有元素相加
`np.all`	测试是否所有元素是真值 （非零数值是真值）
`np.any`	测试是否任意元素是真值（非零数值是真值）
`np.count_nonzero`	计算非零元素的数量

每个这些函数接受字符串数组作为参数，并返回数组。

函数	描述
`np.char.lower`	将每个元素变成小写
`np.char.upper`	将每个元素变成大写
`np.char.strip`	移除每个元素开头或末尾的空格
`np.char.isalpha`	每个元素是否只含有字母（没有数字或者符号）
`np.char.isnumeric`	每个元素是否只含有数字（没有字母）

每个这些函数接受字符串数组和一个搜索字符串。

函数	描述
np.char.count	在数组的元素中，计算搜索字符串的出现次数
np.char.find	在每个元素中，搜索字符串的首次出现位置
np.char.rfind	在每个元素中，搜索字符串的最后一次出现位置
np.char.startswith	每个字符串是否以搜索字符串起始

范围

范围是一个数组，按照递增或递减的顺序排列，每个元素按照一定的间隔分开。 范围在很多情况下非常有用，所以值得了解它们。

范围使用 `np.arange` 函数来定义，该函数接受一个，两个或三个参数：起始值，终止值和“步长”。

如果将一个参数传递给 `np.arange` ，那么它将成为终止值，其中 `start = 0` ， `step = 1` 。 两个参数提供了起始值和终止值， `step = 1` 。 三个参数明确地提供了起始值，终止值和步长。

范围始终包含其 `start` 值，但不包括其 `end` 值。 它按照 `step` 计数，并在到达 `end` 之前停止。

```
np.arange(end): An array starting with 0 of increasing consecutive integers, stopping
before end.
np.arange(5)
array([0, 1, 2, 3, 4])
```

要注意，数值从 `0` 起始，并仅仅增加到 `4` ，并不是 `5` 。

```
np.arange(start, end): An array of consecutive increasing integers from start, stoppin
g before end.
np.arange(3, 9)
array([3, 4, 5, 6, 7, 8])
np.arange(start, end, step): A range with a difference of step between each pair of co
nsecutive values, starting from start and stopping before end.
np.arange(3, 30, 5)
array([ 3,  8, 13, 18, 23, 28])
```

这个数组从 `3` 起始，增加了步长 `5` 变成 `8` ，然后增加步长 `5` 变成 `13` ，以此类推。

当你指定步长时，起始值、终止值和步长可正可负，可以是整数也可以是分数。

```
np.arange(1.5, -2, -0.5)
array([ 1.5,  1. ,  0.5,  0. , -0.5, -1. , -1.5])
```

示例：莱布尼茨的 π 公式

伟大的德国数学家和哲学家戈特弗里德·威廉·莱布尼茨（Gottfried Wilhelm Leibniz，1646 ~ 1716年）发现了一个简单分数的无穷和。 公式是：

$$\pi = 2 \cdot \left(\frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \dots \right)$$

虽然需要一些数学来确定它，但我们可以用数组来说服我们自己，公式是有效的。 让我们计算莱布尼茨的无穷和的前 5000 个项，看它是否接近 π。

我们将计算这个有限的总和，首先加上所有的正项，然后减去所有负项的和 [1]：

$$\pi \approx 2 \cdot \left(\frac{2}{1} \cdot \frac{4}{3} \cdot \frac{6}{5} \dots \frac{1,000,000}{999999} \right) \cdot \left(\frac{2}{3} \cdot \frac{4}{5} \cdot \frac{6}{7} \dots \frac{1,000,000}{1,000,001} \right)$$

> [1] 令人惊讶的是，当我们将无限多个分数相加时，顺序可能很重要。但是我们对 π 的近似只使用了大量的数量有限的分数，所以可以按照任何方便的顺序，将这些项相加。

和中的正项的分母是 `1, 5, 9` ，以此类推。数组 `by_four_to_20` 包含 `17` 之前的这些数。

```
by_four_to_20 = np.arange(1, 20, 4)
by_four_to_20
array([ 1,  5,  9, 13, 17])
```

为了获得 π 的准确近似，我们使用更长的数组 `positive_term_denominators` 。

```
positive_term_denominators = np.arange(1, 10000, 4)
positive_term_denominators
array([   1,     5,     9, ..., 9989, 9993, 9997])
```

我们实际打算加起来的正项，就是一除以这些分母。

```
positive_terms = 1 / positive_term_denominators
```

负向的分母是`3, 7, 11`，以此类推。这个数组就是`positive_term_denominators`加二。

```
negative_terms = 1 / (positive_term_denominators + 2)
```

整体的和是：

```
4 * ( sum(positive_terms) - sum(negative_terms) )
3.1413926535917955
```

这非常接近于`π = 3.14159...`。莱布尼茨公式看起来不错。

五、表格

> 原文：Tables
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

表格是表示数据集的基本对象类型。 表格可以用两种方式查看：

- 具名列的序列，每列都描述数据集中所有条目的一个方面，或者
- 行的序列，每行都包含数据集中单个条目的所有信息。

为了使用表格，导入所有称为 `datascience` 的模块，这是为这篇文章创建的模块。

```
from datascience import *
```

空表格可以使用 `Table` 创建。空表格是实用的，因为他可以扩展来包含新行和新列。

```
Table()
```

表格上的 `with_columns` 方法使用带有附加标签的列，构造一个新表。 表格的每一列都是一个数组。 为了将一个新列添加到表中，请使用标签和数组调用 `with_columns` 。（ `with_column` 方法具有相同的效果。）

下面，我们用一个没有列的空表开始每个例子。

```
Table().with_columns('Number of petals', make_array(8, 34, 5))
```

Number of petals
8
34
5

为了添加两个（或更多）新列，请为每列提供一个数组和标签。 所有列必须具有相同的长度，否则会发生错误。

```
Table().with_columns(
    'Number of petals', make_array(8, 34, 5),
    'Name', make_array('lotus', 'sunflower', 'rose')
)
```

Number of petals	Name
8	lotus
34	sunflower
5	rose

我们可以给这个表格一个名词，之后使用另外一列扩展表格。

```
flowers = Table().with_columns(
    'Number of petals', make_array(8, 34, 5),
    'Name', make_array('lotus', 'sunflower', 'rose')
)

flowers.with_columns(
    'Color', make_array('pink', 'yellow', 'red')
)
```

Number of petals	Name	Color
8	lotus	pink
34	sunflower	yellow
5	rose	red

`with_columns` 方法每次调用时，都会创建一个新表，所以原始表不受影响。 例如，表 `an_example` 仍然只有它创建时的两列。

```
flowers
```

Number of petals	Name
8	lotus
34	sunflower
5	rose

通过这种方式创建表涉及大量的输入。 如果数据已经输入到某个地方，通常可以使用 Python 将其读入表格中，而不是逐个单元格地输入。

通常，表格从包含逗号分隔值的文件创建。这些文件被称为 CSV 文件。

下面，我们使用 `Table` 的 `read_table` 方法，来读取一个 CSV 文件，它包含了一些数据，Minard 在他的拿破仑的俄罗斯战役的图片中使用。 数据放在名为 `minard` 的表中。

```
minard = Table.read_table('minard.csv')
minard
```

Longitude	Latitude	City	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

我们将使用这个小的表格来演示一些有用的表格方法。 然后，我们将使用这些相同的方法，并在更大的数据表上开发其他方法。

表格的大小

`num_columns` 方法提供了表中的列数量，`num_rows` 是行数量。

```
minard.num_columns
5
minard.num_rows
8
```

列标签

`labels` 方法可以用来列出所有列的标签。 对于 `minard` ，并不是特别有用，但是对于那些非常大的表格，并不是所有的列都在屏幕上可见。

```
minard.labels
('Longitude', 'Latitude', 'City', 'Direction', 'Survivors')
```

我们使用 `relabeled` 修改列标签。这会创建新的表格，并保留 `minard` 不变。

```
minard.relabeled('City', 'City Name')
```

Longitude	Latitude	City Name	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

但是，这个方法并不修改原始表。

```
minard
```

Longitude	Latitude	City	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

常见的模式时将原始名称 `minard` 赋给新的表，以便 `minard` 未来的所有使用，都会引用修改标签的表格。

```
minard = minard.relabeled('City', 'City Name')
minard
```

Longitude	Latitude	City Name	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

访问列中的数据

我们可以使用列标签来访问列中的数据数组。

```
minard.column('Survivors')
array([145000, 140000, 127100, 100000,  55000,  24000,  20000,  12000])
```

五列的下标分别为 `0, 1, 2, 3, 4` 。`Survivors` 列也可以使用列下标来访问。

```
minard.column(4)
array([145000, 140000, 127100, 100000,  55000,  24000,  20000,  12000])
```

数组中的八个条目下标为 `0, 1, 2, ..., 7` 。列中的条目可以使用 `item` 访问，就像任何数组那样。

```
minard.column(4).item(0)
145000
minard.column(4).item(5)
24000
```

处理列中的数据

因为列是数组，所以我们可以使用数组操作来探索新的信息。 例如，我们可以创建一个新列，其中包含 Smolensk 之后每个城市的所有幸存者的百分比。

```
initial = minard.column('Survivors').item(0)
minard = minard.with_columns(
    'Percent Surviving', minard.column('Survivors')/initial
)
minard
```

Longitude	Latitude	City Name	Direction	Survivors	Percent Surviving
32	54.8	Smolensk	Advance	145000	100.00%
33.2	54.9	Dorogobouge	Advance	140000	96.55%
34.4	55.5	Chjat	Advance	127100	87.66%
37.6	55.8	Moscou	Advance	100000	68.97%
34.3	55.2	Wixma	Retreat	55000	37.93%
32	54.6	Smolensk	Retreat	24000	16.55%
30.4	54.4	Orscha	Retreat	20000	13.79%
26.8	54.3	Moiodexno	Retreat	12000	8.28%

要使新列中的比例显示为百分比，我们可以使用选项 `PercentFormatter` 调用 `set_format` 方法。 `set_format` 方法接受 `Formatter` 对象，存在日期（ `DateFormatter` ），货币（ `CurrencyFormatter` ），数字和百分比。

```
minard.set_format('Percent Surviving', PercentFormatter)
```

Longitude	Latitude	City Name	Direction	Survivors	Percent Surviving
32	54.8	Smolensk	Advance	145000	100.00%
33.2	54.9	Dorogobouge	Advance	140000	96.55%
34.4	55.5	Chjat	Advance	127100	87.66%
37.6	55.8	Moscou	Advance	100000	68.97%
34.3	55.2	Wixma	Retreat	55000	37.93%
32	54.6	Smolensk	Retreat	24000	16.55%
30.4	54.4	Orscha	Retreat	20000	13.79%
26.8	54.3	Moiodexno	Retreat	12000	8.28%

选择列的集合

`select` 方法创建一个新表，仅仅包含指定的列。

```
minard.select('Longitude', 'Latitude')
```

Longitude	Latitude
32	54.8
33.2	54.9
34.4	55.5
37.6	55.8
34.3	55.2
32	54.6
30.4	54.4
26.8	54.3

使用列索引而不是标签，也可以执行相同选择。

```
minard.select(0, 1)
```

Longitude	Latitude
32	54.8
33.2	54.9
34.4	55.5
37.6	55.8
34.3	55.2
32	54.6
30.4	54.4
26.8	54.3

`select` 的结果是个新表，即使当你选择一列时也是这样。

```
minard.select('Survivors')
```

Survivors
145000
140000
127100
100000
55000
24000
20000
12000

要注意结果是个表格，不像`column`的结果，它是个数组。

```
minard.column('Survivors')
array([145000, 140000, 127100, 100000,  55000,  24000,  20000,  12000])
```

另一种创建新表，包含列集合的方式，是`drop`你不想要的列。

```
minard.drop('Longitude', 'Latitude', 'Direction')
```

City Name	Survivors	Percent Surviving
Smolensk	145000	100.00%
Dorogobouge	140000	96.55%
Chjat	127100	87.66%
Moscou	100000	68.97%
Wixma	55000	37.93%
Smolensk	24000	16.55%
Orscha	20000	13.79%
Moiodexno	12000	8.28%

`select`和`drop`都不修改原始表格。相反，他们创建了共享相同数据的新小型表格。保留的原始表格是实用的！你可以生成多个不同的表格，只考虑某些列，而不用担心会互相影响。

```
minard
```

Longitude	Latitude	City Name	Direction	Survivors	Percent Surviving
32	54.8	Smolensk	Advance	145000	100.00%
33.2	54.9	Dorogobouge	Advance	140000	96.55%
34.4	55.5	Chjat	Advance	127100	87.66%
37.6	55.8	Moscou	Advance	100000	68.97%
34.3	55.2	Wixma	Retreat	55000	37.93%
32	54.6	Smolensk	Retreat	24000	16.55%
30.4	54.4	Orscha	Retreat	20000	13.79%
26.8	54.3	Moiodexno	Retreat	12000	8.28%

我们用过的所有方法都可以用在任何表格上。

对行排序

CNN 在 2016 年 3 月报道说："NBA 是全球薪水最高的职业体育联盟。" `nba_salaries` 包含了 2015~2016 年间所有 NBA 球员的薪水。

每行表示一个球员。列为：

列标签	描述
`PLAYER`	球员名称
`POSITION`	球员在队里的位置
`TEAM`	队的明确
`'15-'16 SALARY`	2015~2016 年的球员薪水，单位是百万美元。

位置代码是 PG（控球后卫），SG（得分后卫），PF（大前锋），SF（小前锋）和 C（中锋）。但接下来的内容并不涉及篮球运动的细节。

第一行显示，亚特兰大老鹰队（Atlanta Hawks）的大前锋保罗·米尔萨普（Paul Millsap）在 2015~2016 年间的薪水接近 1870 万美元。

```
# This table can be found online: https://www.statcrunch.com/app/index.php?dataid=1843
341
nba_salaries = Table.read_table('nba_salaries.csv')
nba_salaries
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4
Mike Scott	PF	Atlanta Hawks	3.33333
Kent Bazemore	SF	Atlanta Hawks	2
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452

（省略了 407 行）

该表包含 417 行，每个球员一行。只显示了 10 行。`show` 方法允许我们指定行数，缺省值（没有指定）是表的所有行。

```
nba_salaries.show(3)
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625

（省略了 414 行）

通过浏览大约 20 行左右，你会看到行按字母顺序排列。也可以使用 `sort` 方法，按球员姓名的字母顺序列出相同的行。`sort` 的参数是列标签或索引。

```
nba_salaries.sort('PLAYER').show(5)
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Aaron Brooks	PG	Chicago Bulls	2.25
Aaron Gordon	PF	Orlando Magic	4.17168
Aaron Harrison	SG	Charlotte Hornets	0.525093
Adreian Payne	PF	Minnesota Timberwolves	1.93884
Al Horford	C	Atlanta Hawks	12

（省略了 412 行）

440/5000 为了检查球员的薪水，如果数据是按薪水排序的话，会更有帮助。

为了实现它，我们首先简化薪水列的标签（只是为了方便），然后用新的标签 `SALARY` 进行排序。

这会按照薪水的升序排列表中的所有行，最低的薪水在最前面。 输出是一个新表，列与原始表格相同，但行是重新排列的。

```
nba = nba_salaries.relabeled("'15-'16 SALARY", 'SALARY')
nba.sort('SALARY')
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Thanasis Antetokounmpo	SF	New York Knicks	0.030888
Jordan McRae	SG	Phoenix Suns	0.049709
Cory Jefferson	PF	Phoenix Suns	0.049709
Elliot Williams	SG	Memphis Grizzlies	0.055722
Orlando Johnson	SG	Phoenix Suns	0.055722
Phil Pressey	PG	Phoenix Suns	0.055722
Keith Appling	PG	Orlando Magic	0.061776
Sean Kilpatrick	SG	Denver Nuggets	0.099418
Erick Green	PG	Utah Jazz	0.099418
Jeff Ayres	PF	Los Angeles Clippers	0.111444

（省略了 407 行）

这些数字有些难以比较，因为这些球员中的一些，在赛季中改变了球队，并从不止一支球队获得了薪水。 只有最后一支球队的薪水出现在表中。 控球后卫菲尔·普莱西（Phil Pressey）在年内从费城搬到了凤凰城，可能会再次转到金州勇士队（Golden State Warriors）。

CNN 的报道是薪酬水平的另一端 - 那些在世界上薪水最高的球员。

为了按照薪水的降序对表格的行排序，我们必须以 `descending=True` 调用 `sort` 函数。

```
nba.sort('SALARY', descending=True)
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Kobe Bryant	SF	Los Angeles Lakers	25
Joe Johnson	SF	Brooklyn Nets	24.8949
LeBron James	SF	Cleveland Cavaliers	22.9705
Carmelo Anthony	SF	New York Knicks	22.875
Dwight Howard	C	Houston Rockets	22.3594
Chris Bosh	PF	Miami Heat	22.1927
Chris Paul	PG	Los Angeles Clippers	21.4687
Kevin Durant	SF	Oklahoma City Thunder	20.1586
Derrick Rose	PG	Chicago Bulls	20.0931
Dwyane Wade	SG	Miami Heat	20

（省略了 407 行）

科比（Kobe Bryant）在湖人队（Lakers）的最后一个赛季是薪水最高的，2500 万美元。 请注意，MVP 斯蒂芬·库里（Stephen Curry）并没有出现在前 10 名之列。他排在后面，我们将在后面看到。

具名参数

这个调用表达式的`descending = True`部分称为具名参数。 调用一个函数或方法时，每个参数都有一个位置和一个名字。 从函数或方法的帮助文本中都可以看出它们。

```
help(nba.sort)
Help on method sort in module datascience.tables:

sort(column_or_label, descending=False, distinct=False) method of datascience.tables.T
able instance
    Return a Table of rows sorted according to the values in a column.

    Args:
        ``column_or_label``: the column whose values are used for sorting.

        ``descending``: if True, sorting will be in descending, rather than
            ascending order.

        ``distinct``: if True, repeated values in ``column_or_label`` will
            be omitted.

    Returns:
        An instance of ``Table`` containing rows sorted based on the values
        in ``column_or_label``.

    >>> marbles = Table().with_columns(
    ...     "Color", make_array("Red", "Green", "Blue", "Red", "Green", "Green"),
    ...     "Shape", make_array("Round", "Rectangular", "Rectangular", "Round", "Rectan
gular", "Round"),
    ...     "Amount", make_array(4, 6, 12, 7, 9, 2),
    ...     "Price", make_array(1.30, 1.30, 2.00, 1.75, 1.40, 1.00))
    >>> marbles
    Color | Shape       | Amount | Price
    Red   | Round       | 4      | 1.3
    Green | Rectangular | 6      | 1.3
    Blue  | Rectangular | 12     | 2
    Red   | Round       | 7      | 1.75
    Green | Rectangular | 9      | 1.4
    Green | Round       | 2      | 1
    >>> marbles.sort("Amount")
    Color | Shape       | Amount | Price
    Green | Round       | 2      | 1
    Red   | Round       | 4      | 1.3
    Green | Rectangular | 6      | 1.3
    Red   | Round       | 7      | 1.75
    Green | Rectangular | 9      | 1.4
    Blue  | Rectangular | 12     | 2
    >>> marbles.sort("Amount", descending = True)
    Color | Shape       | Amount | Price
    Blue  | Rectangular | 12     | 2
    Green | Rectangular | 9      | 1.4
    Red   | Round       | 7      | 1.75
    Green | Rectangular | 6      | 1.3
    Red   | Round       | 4      | 1.3
    Green | Round       | 2      | 1
    >>> marbles.sort(3) # the Price column
    Color | Shape       | Amount | Price
    Green | Round       | 2      | 1
    Red   | Round       | 4      | 1.3
    Green | Rectangular | 6      | 1.3
    Green | Rectangular | 9      | 1.4
    Red   | Round       | 7      | 1.75
    Blue  | Rectangular | 12     | 2
    >>> marbles.sort(3, distinct = True)
    Color | Shape       | Amount | Price
    Green | Round       | 2      | 1
    Red   | Round       | 4      | 1.3
    Green | Rectangular | 9      | 1.4
    Red   | Round       | 7      | 1.75
    Blue  | Rectangular | 12     | 2
```

在 `help` 文本的最上面，出现了 `sort` 方法的签名。

```
sort(column_or_label, descending=False, distinct=False)
```

这描述了`sort`的三个参数的位置，名称和默认值。 调用此方法时，可以使用位置参数或具名参数，因此以下三个调用完全相同。

```
sort('SALARY', True)
sort('SALARY', descending=True)
sort(column_or_label='SALARY', descending=True)
```

当一个参数只是`True`或`False`时，包含参数名称是实用的约定，以便更明显地说明参数值的含义。

行的选取

通常，我们只想提取那些行，它们对应具有特定特征的条目。 例如，我们可能只需要对应勇士的行，或者获得超过一千万美元的球员。 或者我们可能只想要薪水前五名的人。

指定行

`Table`的方法就是干这个的 - 它需要一组指定的行。 它的参数是行索引或索引数组，它创建一个只包含这些行的新表。

例如，如果我们只想要`nba`的第一行，我们可以这样使用`take`。

```
nba
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4
Mike Scott	PF	Atlanta Hawks	3.33333
Kent Bazemore	SF	Atlanta Hawks	2
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452

（省略了 407 行）

```
nba.take(0)
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717

这是一个新表，只拥有我们指定的单个行。

通过指定一系列索引作为参数，我们还可以获得第四，第五和第六行。

```
nba.take(np.arange(3, 6))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4

如果我们想要前五个最高薪球员的表格，我们可以先按薪水排序，然后取前五行：

```
nba.sort('SALARY', descending=True).take(np.arange(5))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Kobe Bryant	SF	Los Angeles Lakers	25
Joe Johnson	SF	Brooklyn Nets	24.8949
LeBron James	SF	Cleveland Cavaliers	22.9705
Carmelo Anthony	SF	New York Knicks	22.875
Dwight Howard	C	Houston Rockets	22.3594

对应指定特征的行

更常见的情况是，我们打算访问一组行中的数据，它们具有某种特征，但是我们并不知道其索引。 例如，我们可能想要所有薪水大于一千万美元的球员的数据，但我们不希望花费时间，对已排序的表中的行进行计数。

`where` 方法可以做到。 它的输出是一个表格，列与原始表格相同，但只有特征出现的行。

`where` 的第一个参数是列标签，列中包含信息，有关某行是否具有我们想要的特征。 如果特征是“薪水超过一千万美元”，那么列就是 `SALARY` 。

`where` 的第二个参数是用于指定特征的方式。 一些例子会使指定的一般方式更容易理解。

在第一个例子中，我们提取了所有薪水超过一千万美元的人的数据。

```
nba.where('SALARY', are.above(10))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Joe Johnson	SF	Brooklyn Nets	24.8949
Thaddeus Young	PF	Brooklyn Nets	11.236
Al Jefferson	C	Charlotte Hornets	13.5
Nicolas Batum	SG	Charlotte Hornets	13.1253
Kemba Walker	PG	Charlotte Hornets	12
Derrick Rose	PG	Chicago Bulls	20.0931
Jimmy Butler	SG	Chicago Bulls	16.4075
Joakim Noah	C	Chicago Bulls	13.4

（省略了 59 行）

`are.above(10)` 的参数的确保了，每个选择的行的 `SALARY` 大于 10。

新的表格有 69 行，相当于 69 个球员的薪水是一千万美元。 按顺序排列这些行使数据更易于分析。 多伦多猛龙队（Toronto Raptors）的德玛尔·德罗赞（DeMar DeRozan）是这个分组（薪水超过一千万美元）中“最穷”的一个。

```
nba.where('SALARY', are.above(10)).sort('SALARY')
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
DeMar DeRozan	SG	Toronto Raptors	10.05
Gerald Wallace	SF	Philadelphia 76ers	10.1059
Luol Deng	SF	Miami Heat	10.1516
Monta Ellis	SG	Indiana Pacers	10.3
Wilson Chandler	SF	Denver Nuggets	10.4494
Brendan Haywood	C	Cleveland Cavaliers	10.5225
Jrue Holiday	PG	New Orleans Pelicans	10.5955
Tyreke Evans	SG	New Orleans Pelicans	10.7346
Marcin Gortat	C	Washington Wizards	11.2174
Thaddeus Young	PF	Brooklyn Nets	11.236

（省略了 59 行）

斯蒂芬·库里（Stephen Curry）挣了多少？ 对于答案，我们必须访问 `PLAYER` 的值等于 `Stephen Curry` 的行。 这是一个只包含一行的表格：

```
nba.where('PLAYER', are.equal_to('Stephen Curry'))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Stephen Curry	PG	Golden State Warriors	11.3708

库里只有不到 1140 万美元。 这是很多钱，但还不到勒布朗·詹姆斯（LeBron James）薪水的一半。 你可以在本节前面的“前 5 名”表中找到薪水，或者你可以在上面的代码中找到它，将 `'Stephen Curry'` 换成 `'LeBron James'` 。

代码中再次使用了 `are` ，但这次是谓词 `equal_to` 而不是上面那个。 因此，例如，你可以得到包含所有的勇士的表格：

```
nba.where('TEAM', are.equal_to('Golden State Warriors')).show()
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Klay Thompson	SG	Golden State Warriors	15.501
Draymond Green	PF	Golden State Warriors	14.2609
Andrew Bogut	C	Golden State Warriors	13.8
Andre Iguodala	SF	Golden State Warriors	11.7105
Stephen Curry	PG	Golden State Warriors	11.3708
Jason Thompson	PF	Golden State Warriors	7.00847
Shaun Livingston	PG	Golden State Warriors	5.54373
Harrison Barnes	SF	Golden State Warriors	3.8734
Marreese Speights	C	Golden State Warriors	3.815
Leandro Barbosa	SG	Golden State Warriors	2.5
Festus Ezeli	C	Golden State Warriors	2.00875
Brandon Rush	SF	Golden State Warriors	1.27096
Kevon Looney	SF	Golden State Warriors	1.13196
Anderson Varejao	PF	Golden State Warriors	0.289755

这部分表格已经按薪水排序，因为原始的表格按薪水排序，列出了同一个球队中球员。 行尾的 `.show()` 确保显示所有行，而不仅仅是前 10 行。

请求某列等于某值的行非常普遍，因此 `are.equal_to` 调用是可选的。 相反，仅仅使用列名和值来调用 `where` 方法，以达到相同的效果。

```
nba.where('TEAM', 'Denver Nuggets')
# equivalent to nba.where('TEAM', are.equal_to('Denver Nuggets'))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Danilo Gallinari	SF	Denver Nuggets	14
Kenneth Faried	PF	Denver Nuggets	11.236
Wilson Chandler	SF	Denver Nuggets	10.4494
JJ Hickson	C	Denver Nuggets	5.6135
Jameer Nelson	PG	Denver Nuggets	4.345
Will Barton	SF	Denver Nuggets	3.53333
Emmanuel Mudiay	PG	Denver Nuggets	3.10224
Darrell Arthur	PF	Denver Nuggets	2.814
Jusuf Nurkic	C	Denver Nuggets	1.842
Joffrey Lauvergne	C	Denver Nuggets	1.70972

（省略了 4 行）

多个属性

通过重复使用 `where`，你可以访问具有多个指定特征的行。 例如，这是一种方法，提取薪水超过一千五百万美元的所有控球后卫。

```
nba.where('POSITION', 'PG').where('SALARY', are.above(15))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Derrick Rose	PG	Chicago Bulls	20.0931
Kyrie Irving	PG	Cleveland Cavaliers	16.4075
Chris Paul	PG	Los Angeles Clippers	21.4687
Russell Westbrook	PG	Oklahoma City Thunder	16.7442
John Wall	PG	Washington Wizards	15.852

一般形式

现在你已经意识到，通过选择具有给定特征的行，来创建新表的一般方法，是使用 `where` 和 `are`，以及适当的条件：

```
original_table_name.where(column_label_string, are.condition)

nba.where('SALARY', are.between(10, 10.3))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Luol Deng	SF	Miami Heat	10.1516
Gerald Wallace	SF	Philadelphia 76ers	10.1059
Danny Green	SG	San Antonio Spurs	10
DeMar DeRozan	SG	Toronto Raptors	10.05

请注意，上面的表格包括赚一千万美元的 Danny Green，而不包括一千三百万美元的 Monta Ellis。 与 Python 中的其他地方一样，范围包括左端但不包括右端。

如果我们指定一个任何行都不满足的条件，我们得到一个带有列标签但没有行的表。

```
nba.where('PLAYER', are.equal_to('Barack Obama'))
```

PLAYER	POSITION	TEAM	SALARY

更多条件

这里有一些谓词，你可能会觉得有用。 请注意，`x` 和 `y` 是数字，`STRING` 是一个字符串，`Z` 是数字或字符串；你必须指定这些，取决于你想要的特征。

谓词	描述
`are.equal_to(Z)`	等于 `Z`
`are.above(x)`	大于 `x`
`are.above_or_equal_to(x)`	大于等于 `x`
`are.below(x)`	小于 `x`
`are.below_or_equal_to(x)`	小于等于 `x`
`are.between(x, y)`	大于等于 `x` ，小于 `y`
`are.strictly_between(x, y)`	大于 `x` ，小于 `y`
`are.between_or_equal_to(x, y)`	大于等于 `x` ，小于等于 `y`
`are.containing(S)`	包含字符串 `S`

你也可以指定任何这些条件的否定，通过在条件前面使用 `.not_` 。

谓词	描述
`are.not_equal_to(Z)`	不等于 `Z`
`are.not_above(x)`	不大于 `x`

以及其他。通常的逻辑规则是适用的，例如，“不大于 `x` ”等价于“小于等于 `x` ”。

我们以一系列示例结束这一节。

`are.containing` 的使用有助于少打一些字。 例如，你可以指定 `Warriors` 而不是 `Golden State Warriors` ：

```
nba.where('TEAM', are.containing('Warriors')).show()
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Klay Thompson	SG	Golden State Warriors	15.501
Draymond Green	PF	Golden State Warriors	14.2609
Andrew Bogut	C	Golden State Warriors	13.8
Andre Iguodala	SF	Golden State Warriors	11.7105
Stephen Curry	PG	Golden State Warriors	11.3708
Jason Thompson	PF	Golden State Warriors	7.00847
Shaun Livingston	PG	Golden State Warriors	5.54373
Harrison Barnes	SF	Golden State Warriors	3.8734
Marreese Speights	C	Golden State Warriors	3.815
Leandro Barbosa	SG	Golden State Warriors	2.5
Festus Ezeli	C	Golden State Warriors	2.00875
Brandon Rush	SF	Golden State Warriors	1.27096
Kevon Looney	SF	Golden State Warriors	1.13196
Anderson Varejao	PF	Golden State Warriors	0.289755

你可以提取所有后卫的数据，包括控球后卫和得分后卫。

```
nba.where('POSITION', are.containing('G'))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452
Jason Richardson	SG	Atlanta Hawks	0.947276
Lamar Patterson	SG	Atlanta Hawks	0.525093
Terran Petteway	SG	Atlanta Hawks	0.525093
Avery Bradley	PG	Boston Celtics	7.73034
Isaiah Thomas	PG	Boston Celtics	6.91287
Marcus Smart	PG	Boston Celtics	3.43104

（省略了 171 行）

你可以获取所有不是克利夫兰骑士队的球员，并且薪水不低于两千万美元：

```
other_than_Cavs = nba.where('TEAM', are.not_equal_to('Cleveland Cavaliers'))
other_than_Cavs.where('SALARY', are.not_below(20))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Joe Johnson	SF	Brooklyn Nets	24.8949
Derrick Rose	PG	Chicago Bulls	20.0931
Dwight Howard	C	Houston Rockets	22.3594
Chris Paul	PG	Los Angeles Clippers	21.4687
Kobe Bryant	SF	Los Angeles Lakers	25
Chris Bosh	PF	Miami Heat	22.1927
Dwyane Wade	SG	Miami Heat	20
Carmelo Anthony	SF	New York Knicks	22.875
Kevin Durant	SF	Oklahoma City Thunder	20.1586

有很多方式可以创建相同的表格。这里是另一种，并且显然你可以想出来更多。

```
other_than_Cavs.where('SALARY', are.above_or_equal_to(20))
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Joe Johnson	SF	Brooklyn Nets	24.8949
Derrick Rose	PG	Chicago Bulls	20.0931
Dwight Howard	C	Houston Rockets	22.3594
Chris Paul	PG	Los Angeles Clippers	21.4687
Kobe Bryant	SF	Los Angeles Lakers	25
Chris Bosh	PF	Miami Heat	22.1927
Dwyane Wade	SG	Miami Heat	20
Carmelo Anthony	SF	New York Knicks	22.875
Kevin Durant	SF	Oklahoma City Thunder	20.1586

你可以看到，`where` 的使用提供了很大的灵活性，来访问你感兴趣的特征。 不要犹豫，尝试它吧！

示例：人口趋势

美国人口的趋势

现在我们做好了处理大量的数据表的准备。 下面的文件包含“美国居民人口的年度估计，按年龄和性别分列”。 请注意，`read_table` 可以直接从 URL 读取数据。

```
# As of Jan 2017, this census file is online here:
data = 'http://www2.census.gov/programs-surveys/popest/datasets/2010-2015/national/asr
h/nc-est2015-agesex-res.csv'

# A local copy can be accessed here in case census.gov moves the file:
# data = 'nc-est2015-agesex-res.csv'

full_census_table = Table.read_table(data)
full_census_table
```

SEX	AGE	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010
0	0	3944153	3944160	3951330
0	1	3978070	3978090	3957888
0	2	4096929	4096939	4090862
0	3	4119040	4119051	4111920
0	4	4063170	4063186	4077551
0	5	4056858	4056872	4064653
0	6	4066381	4066412	4073013
0	7	4030579	4030594	4043046
0	8	4046486	4046497	4025604
0	9	4148353	4148369	4125415

（已省略 296 行）

只显示了表格的前 10 行。稍后我们将看到如何显示整个表格；但是，这通常不适用于大型表格。

表格的描述一起出现。 `SEX` 列包含数字代码： `0` 代表总体， `1代` 表男性， `2` 代表女性。 `AGE` 列包含完整年份为单位的年龄，但特殊值 `999` 是人口的总和。 其余的列包含美国人口的估计。

通常，公共表格将包含更多的信息，不仅仅是特定调查或分析所需的信息。 在这种情况下，我们假设我们只对 2010 年到 2014 年的人口变化感兴趣。让我们选择相关的列。

```
partial_census_table = full_census_table.select('SEX', 'AGE', 'POPESTIMATE2010', 'POPE
STIMATE2014')
partial_census_table
```

SEX	AGE	POPESTIMATE2010	POPESTIMATE2014
0	0	3951330	3949775
0	1	3957888	3949776
0	2	4090862	3959664
0	3	4111920	4007079
0	4	4077551	4005716
0	5	4064653	4006900
0	6	4073013	4135930
0	7	4043046	4155326
0	8	4025604	4120903
0	9	4125415	4108349

（已省略 296 行）

我们也可以简化所选列的标签。

```
us_pop = partial_census_table.relabeled('POPESTIMATE2010', '2010').relabeled('POPESTIM
ATE2014', '2014')
us_pop
```

SEX	AGE	2010	2014
0	0	3951330	3949775
0	1	3957888	3949776
0	2	4090862	3959664
0	3	4111920	4007079
0	4	4077551	4005716
0	5	4064653	4006900
0	6	4073013	4135930
0	7	4043046	4155326
0	8	4025604	4120903
0	9	4125415	4108349

（已省略 296 行）

我们现在有了一个易于使用的表格。 表中的每一列都是一个等长的数组，因此列可以使用算术进行组合。 这是 2010 年至 2014 年的人口变化。

```
us_pop.column('2014') - us_pop.column('2010')
array([  -1555,    -8112, -131198, ...,     6443,    12950, 4693244])
```

让我们使用包含这些变化的列来扩展 `us_pop`，一列是绝对数值，另一列是相对于 2010 年数值的百分比。

```
change = us_pop.column('2014') - us_pop.column('2010')
census = us_pop.with_columns(
    'Change', change,
    'Percent Change', change/us_pop.column('2010')
)
census.set_format('Percent Change', PercentFormatter)
```

SEX	AGE	2010	2014	Change	Percent Change
0	0	3951330	3949775	-1555	-0.04%
0	1	3957888	3949776	-8112	-0.20%
0	2	4090862	3959664	-131198	-3.21%
0	3	4111920	4007079	-104841	-2.55%
0	4	4077551	4005716	-71835	-1.76%
0	5	4064653	4006900	-57753	-1.42%
0	6	4073013	4135930	62917	1.54%
0	7	4043046	4155326	112280	2.78%
0	8	4025604	4120903	95299	2.37%
0	9	4125415	4108349	-17066	-0.41%

（已省略 296 行）

将数据排序。让我们按照人口绝对变化的降序排序表格。

```
census.sort('Change', descending=True)
```

SEX	AGE	2010	2014	Change	Percent Change
0	999	309346863	318907401	9560538	3.09%
1	999	152088043	156955337	4867294	3.20%
2	999	157258820	161952064	4693244	2.98%
0	67	2693707	3485241	791534	29.38%
0	64	2706055	3487559	781504	28.88%
0	66	2621335	3347060	725725	27.69%
0	65	2678525	3382824	704299	26.29%
0	71	1953607	2519705	566098	28.98%
0	34	3822189	4364748	542559	14.19%
0	23	4217228	4702156	484928	11.50%

毫不奇怪，排序后表格的第一行对应整个人口：所有年龄和性别的分组。 从 2010 年到 2014 年，美国人口增加了约 950 万人，仅为 3%。

接下来的两行分别对应所有的男性和所有的女性。 以绝对数量和百分比来衡量，男性人口的增长高于女性人口。 百分比变化都在 3% 左右。

现在看看接下来的几行。 百分比变化从总人口的 3% 左右，上升到 60 年代末和 70 年代初的近 30%。 这个惊人的变化称为美国的老龄化。

到目前为止，2014 年 64~67 岁年龄段的绝对变化最大。什么可以解释这一大幅增长的原因？我们可以通过考察相关分组的出生年份，来探索这个问题。

那些 2010 年在 64~67 岁年龄段的人，在 1943 年到 1946 年间出生。珍珠港的袭击是在 1941 年底，美军在 1942 年发动了一场大规模战争，结束于 1945 年。

2014 年 64 岁到 67 岁的人生于 1947 年到 1950 年，是美国二战后的生育高峰。

战后的生育高峰，是我们观察到的巨大变化的主要原因。

示例：性别趋势

性别比例的趋势

我们现在拥有了足够的编码技能，足以检查美国人群中的特征和趋势。在这个例子中，我们将查看不同年龄的男性和女性的分布情况。我们将继续使用上一节中的 `us_pop` 表。

```
us_pop
```

SEX	AGE	2010	2014
0	0	3951330	3949775
0	1	3957888	3949776
0	2	4090862	3959664
0	3	4111920	4007079
0	4	4077551	4005716
0	5	4064653	4006900
0	6	4073013	4135930
0	7	4043046	4155326
0	8	4025604	4120903
0	9	4125415	4108349

（已省略 296 行）

我们从之前对这个数据集的检查得知，表格的描述一起出现。 它提醒了表中包含的内容。

每一行表示一个年龄。`SEX` 列包含数字代码：`0` 代表总数，`1` 代表男性，`2` 代表女性。 年龄栏包含以完整年份为单位的年龄，但特殊值 `999` 代表整个人口，不论年龄是什么。其余的列包含美国人口的估计。

理解 `AGE=100`

作为一个初步的例子，我们来解释一下表中最后的年龄的数据，其中年龄是 `100` 岁。下面的代码提取了男性和女性（性别代码 `0`）的组合分组，年龄最大的行。

```
us_pop.where('SEX', are.equal_to(0)).where('AGE', are.between(97, 101))
```

SEX	AGE	2010	2014
0	97	68893	83089
0	98	47037	59726
0	99	32178	41468
0	100	54410	71626

不足为奇的是，年龄越大，人数越少，例如 99 岁的人数少于 98 岁。

然而，令人吃惊的是，100 岁的数值比 99 岁的数值要大得多。仔细查看文件，这是因为人口普查局使用 100 作为 100 或岁以上的每个人的代码。

年龄为 100 岁的人不仅仅代表 100 岁的人，还包括年龄在 100 岁以上的人。这就是为什么那一行的数字大于 99 岁的人的数字。

男性和女性的整体比例

我们现在开始考察 2014 年的性别比例。首先，我们一起来看看所有的年龄。 请记住，这意味着查看 `AGE` 编码为 `999` 的行。 `all_ages` 表包含此信息。其中有三行：一个是两种性别总体，一个是男性（ `SEX` 代码为 `1` ），一个是女性（ `SEX` 代码为 `2` ）。

```
us_pop_2014 = us_pop.drop('2010')
all_ages = us_pop_2014.where('AGE', are.equal_to(999))
all_ages
```

SEX	AGE	2014
0	999	318907401
1	999	156955337
2	999	161952064

`all_ages` 的第 0 行包含两年中每年的美国总人口。2014 年，美国人口刚刚少于 3.19 亿。

第 1 行包含男性的计数，女性是第 2 行。 比较这两行可以看到，在 2014 年，美国的女性比男性多。

第 1 行和第 2 行的人口数加起来为第 0 行的总人口数。

为了与其他数量进行比较，我们需要将这些数字转换为总人口中的百分比。 让我们访问 2014 年的总数并命名。 然后，我们将显示带有比例列的人口表格。 与我们先前观察到的，女性人数多于男性的情况一致，2014 年约有 50.8% 的人口是女性，两年的每年中，约有 49.2% 的人口是男性。

```
pop_2014 = all_ages.column('2014').item(0)
all_ages.with_column(
    'Proportion', all_ages.column('2014')/pop_2014
).set_format('Proportion', PercentFormatter)
```

SEX	AGE	2014	Proportion
0	999	318907401	100.00%
1	999	156955337	49.22%
2	999	161952064	50.78%

新生儿中男孩和女孩的比例

但是，当我们查看婴儿时，情况正好相反。 让我们将婴儿定义为还没有完整一年的人，对应年龄为 0 的行。这里是他们的人口数量。 你可以看到男婴比女婴多。

```
infants = us_pop_2014.where('AGE', are.equal_to(0))
infants
```

SEX	AGE	2014
0	0	3949775
1	0	2020326
2	0	1929449

像以前一样，我们可以将这些数字转换成婴儿总数中的百分比。 所得表格显示，2014 年，美国超过 51% 的婴儿是男性。

```
infants_2014 = infants.column('2014').item(0)
infants.with_column(
    'Proportion', infants.column('2014')/infants_2014
).set_format('Proportion', PercentFormatter)
```

SEX	AGE	2014	Proportion
0	0	3949775	100.00%
1	0	2020326	51.15%
2	0	1929449	48.85%

事实上，长期以来，新生儿中男孩的比例略高于 1/2。 这个原因还没有得到彻底的理解，科学家们还在努力。

每个年龄的男女比例

我们已经看到，虽然男婴比女婴多，但女性总数比男性多。 所以很显然，性别之间的分隔在不同年龄之间必须有所不同。

为了研究这个变化，我们将女性和男性的数据分开，并消除所有年龄的组合，年龄编码为 999 的行。

`females` 和 `male` 表格分别包含两个性别的数据。

```
females_all_rows = us_pop_2014.where('SEX', are.equal_to(2))
females = females_all_rows.where('AGE', are.not_equal_to(999))
females
```

SEX	AGE	2014
2	0	1929449
2	1	1931375
2	2	1935991
2	3	1957483
2	4	1961199
2	5	1962561
2	6	2024870
2	7	2032494
2	8	2015285
2	9	2010659

（省略了 91 行）

```
males_all_rows = us_pop_2014.where('SEX', are.equal_to(1))
males = males_all_rows.where('AGE', are.not_equal_to(999))
males
```

SEX	AGE	2014
1	0	2020326
1	1	2018401
1	2	2023673
1	3	2049596
1	4	2044517
1	5	2044339
1	6	2111060
1	7	2122832
1	8	2105618
1	9	2097690

（省略了 91 行）

现在的计划是，比较两年中每一年的，每个年龄的女性人数和男性人数。 数组和表格的方法为我们提供了直接的方式。 这两个表格中，每个年龄都有一行。

```
males.column('AGE')
array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
        26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,
        39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
        52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,
        65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,
        78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,
        91,  92,  93,  94,  95,  96,  97,  98,  99, 100])
females.column('AGE')
array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
        26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,
        39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
        52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,
        65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,
        78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,
        91,  92,  93,  94,  95,  96,  97,  98,  99, 100])
```

对于任何特定年龄，我们都可以通过将女性人数除以男性人数，获的女性：男性性别比例。为了一步完成它，我们可以使用列来提取女性数量的数组，和相应的男性数量的数组，然后简单地将一个数组除以另一个数组。 逐元素相除将为所有年份创建性别比例的数组。

```
ratios = Table().with_columns(
    'AGE', females.column('AGE'),
    '2014 F:M RATIO', females.column('2014')/males.column('2014')
)
ratios
```

AGE	2014 F:M RATIO
0	0.955019
1	0.956884
2	0.956672
3	0.955058
4	0.959248
5	0.959998
6	0.959172
7	0.957445
8	0.957099
9	0.958511

（省略了 91 行）

从输出中可以看出，九岁以下儿童的比例都在 0.96 左右。 当男女比例小于 1 时，女性比男性少。 因此，我们所看到的是，在 0~9 岁的年龄段中，女孩比男孩少。此外，在每个年龄中，每 100 个男孩大约对应 96 个女孩。

那么人口中女性的整体比例为什么高于男性呢？

当我们检查年龄的另一端时，会发现一些非同寻常的事情。 以下是 75 岁以上的男女比例。

```
ratios.where('AGE', are.above(75)).show()
```

AGE	2014 F:M RATIO
76	1.23487
77	1.25797
78	1.28244
79	1.31627
80	1.34138
81	1.37967
82	1.41932
83	1.46552
84	1.52048
85	1.5756
86	1.65096
87	1.72172
88	1.81223
89	1.91837
90	2.01263
91	2.09488
92	2.2299
93	2.33359
94	2.52285
95	2.67253
96	2.87998
97	3.09104
98	3.41826
99	3.63278
100	4.25966

不仅所有这些比例大于 1，在所有这些年龄段中，女性比男性多，其中许多比例大于 1。

- 在 89 岁和 90 岁中，比例接近 2，这意味着 2014 年这些年龄的女性约为男性的两倍。
- 在 98 岁和 99 岁中，女性约为男性的 3.5 至 4 倍。

如果你想知道有多少高龄的人，你可以使用 Python 来发现：

```
males.where('AGE', are.between(98, 100))
```

SEX	AGE	2014
1	98	13518
1	99	8951

```
females.where('AGE', are.between(98, 100))
```

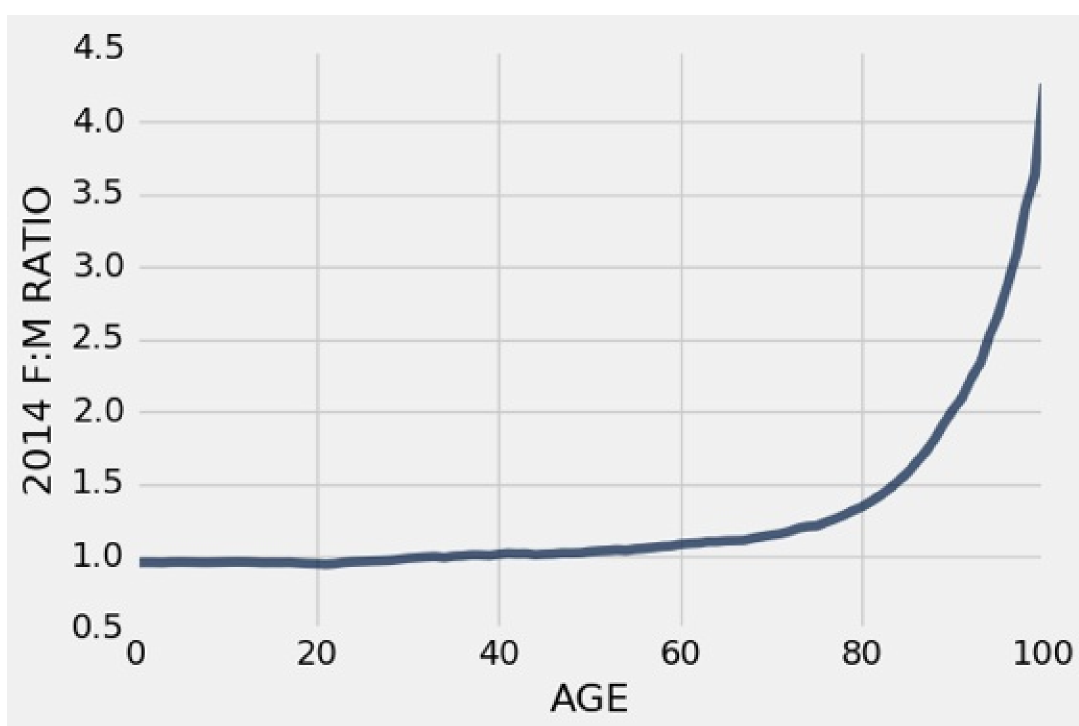
SEX	AGE	2014
2	98	46208
2	99	32517

下图展示了年龄相关的性别比率。 蓝色曲线显示 2014 年的比例与年龄。

从 0 岁到 60 岁，这个比例差不多是 1（表示男性和女性差不多相等），但从 65 岁开始，比例开始急剧上升（女性多于男性）。

美国女性人数多于男性，部分原因是老年妇女的显着的性别不平衡。

```
ratios.plot('AGE')
```



六、可视化

> 原文：Visualization
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

表格是一种组织和可视化数据的强大方式。然而，无论数据如何组织，数字的大型表格可能难以解释。 有时解释图片比数字容易得多。

在本章中，我们将开发一些数据分析的基本图形方法。 我们的数据源是互联网电影数据库（IMDB），这是一个在线数据库，包含电影，电视节目，和视频游戏等信息。Box Office Mojo 网站提供了许多 IMDB 数据摘要，我们已经采用了其中一些。 我们也使用了 The Numbers 的数据摘要，这个网站的口号是“数据和电影业务的相遇之处”。

散点图和线形图

`actors` 表包含好莱坞的男性和女性演员的数据。 其中的列是：

列	内容
Actor	演员名称
Total Gross	演员所有电影的国内票房总收入（百万美元）
Number of Movies	演员所演的电影数量
Average per Movie	总收入除以电影数量
#1 Movie	演员所演的票房最高的电影
Gross	演员的 #1 电影的国内票房总收入（百万美元）

在总票房的计算中，数据的制表人没有包括一些电影，其中演员是客串角色或陈述角色，没有太多的登场时间。

这个表格有 50 行，对应着 50 个最顶级的演员。 这个表已经按照 `Total Gross` 排序了，所以很容易看出，`Harrison Ford` 是最棒的演员。 总的来说，他的电影的国内票房收入比其他演员的电影多。

```
actors = Table.read_table('actors.csv')
actors
```

Actor	Total Gross	Number of Movies	Average per Movie	#1 Movie	Gross
Harrison Ford	4871.7	41	118.8	Star Wars: The Force Awakens	936.7
Samuel L. Jackson	4772.8	69	69.2	The Avengers	623.4
Morgan Freeman	4468.3	61	73.3	The Dark Knight	534.9
Tom Hanks	4340.8	44	98.7	Toy Story 3	415
Robert Downey, Jr.	3947.3	53	74.5	The Avengers	623.4
Eddie Murphy	3810.4	38	100.3	Shrek 2	441.2
Tom Cruise	3587.2	36	99.6	War of the Worlds	234.3
Johnny Depp	3368.6	45	74.9	Dead Man's Chest	423.3
Michael Caine	3351.5	58	57.8	The Dark Knight	534.9
Scarlett Johansson	3341.2	37	90.3	The Avengers	623.4

（已省略 40 行）

术语。变量是我们称之为“特征”的东西的正式名称，比如`'number of movies'`。 术语“变量”强调了，对于不同的个体，这个特征可以有不同的值 - 演员所演电影的数量因人而异。

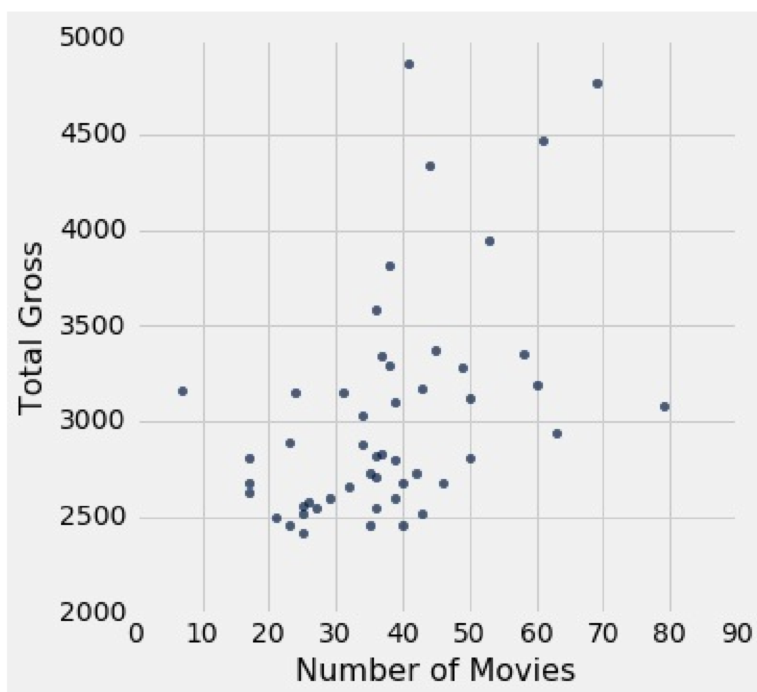
拥有数值的变量（如`'number of movies'`或`'average gross receipts per movie'`）的变量称为定量或数值变量。

散点图

散点图展示两个数值变量之间的关系。 在前面的章节中，我们看到了一个散点图的例子，我们看了两个经典小说的时间段和角色数量。

`Table`的`scatter`方法绘制一个散点图，由表格的每一行组成。它的第一个参数是要在横轴上绘制的列标签，第二个参数是纵轴上的列标签。

```
actors.scatter('Number of Movies', 'Total Gross')
```



散点图包含 50 个点，表中的每个演员为一个。 一般来说，你可以看到它向上倾斜。 一个演员的电影越多，所有这些电影的总收入就越多。

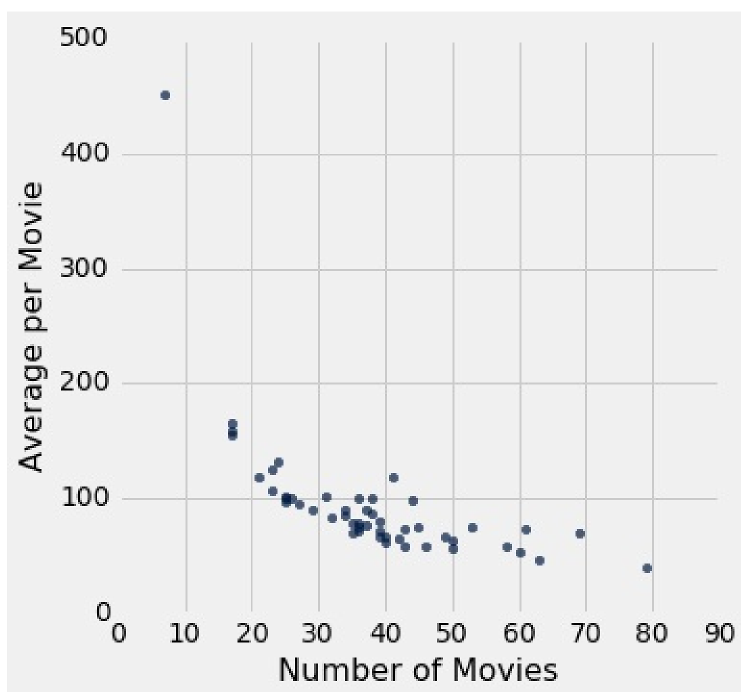
在形式上，我们说图表显示了变量之间的关联，并且关联是正的：一个变量的高值往往与另一个变量的高值相关联，而低值也是一样，通常情况下。

当然有一些变化。 一些演员有很多电影，但总收入中等。 其他人电影数量中等，但收入很高。正相关只是一个大体趋势的叙述。

在课程后面，我们将学习如何量化关联。目前，我们只是定性地思考。

现在我们已经探索了电影的数量与总收入的关系，让我们把注意力转向它与每部电影的平均收入的关系。

```
actors.scatter('Number of Movies', 'Average per Movie')
```



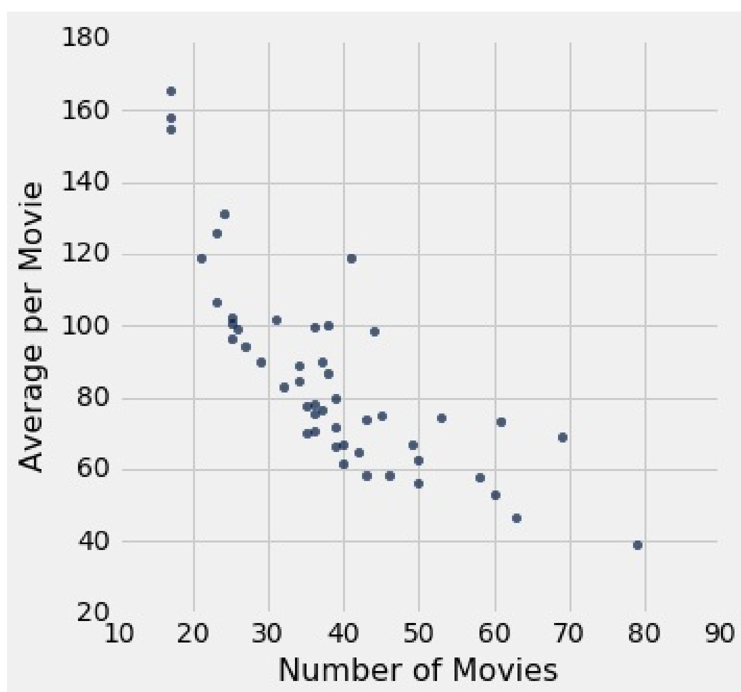
这是一个截然不同的情况，并表现出负相关。 一般来说，演员的电影数量越多，每部电影的平均收入就越少。

另外，有一个点是非常高的，在绘图的左边。 它对应于一个电影数量很少，每部电影平均值很高的演员。 这个点是异常的。 它位于数据的一般范围之外。 事实上，这与绘图中的其他所有点相差甚远。

我们将通过查看绘图的左右两端的点，来进一步检查负相关。

对于右端，我们通过查看没有异常值的部分来放大图的主体。

```
no_outlier = actors.where('Number of Movies', are.above(10))
no_outlier.scatter('Number of Movies', 'Average per Movie')
```



负相关仍然清晰可见。 让我们找出一些演员，对应位于绘图右侧的点，这里电影数量较多：

```
actors.where('Number of Movies', are.above(60))
```

Actor	Total Gross	Number of Movies	Average per Movie	#1 Movie	Gross
Samuel L. Jackson	4772.8	69	69.2	The Avengers	623.4
Morgan Freeman	4468.3	61	73.3	The Dark Knight	534.9
Robert DeNiro	3081.3	79	39	Meet the Fockers	279.3
Liam Neeson	2942.7	63	46.7	The Phantom Menace	474.5

伟大的演员罗伯特·德尼罗（Robert DeNiro）拥有最高的电影数量和最低的每部电影的平均收入。 其他优秀的演员在不远处的点，但德尼罗在极远处。

为了理解负相关，请注意，演员所演的电影越多，在风格，流派和票房方片，这些电影变化就越大。 例如，一个演员可能会出现在一些高收入的动作电影或喜剧中（如 Meet Fockers），也可能是优秀但不会吸引大量人群的小众电影。 因此，演员的每部电影的平均收入值可能相对较低。

为了从不同的角度来看待这个观点，现在让我们来看看这个异常点。

```
actors.where('Number of Movies', are.below(10))
```

Actor	Total Gross	Number of Movies	Average per Movie	#1 Movie	Gross
Anthony Daniels	3162.9	7	451.8	Star Wars: The Force Awakens	936.7

作为一名演员，安东尼·丹尼尔斯（Anthony Daniels）可能没有罗伯特·德尼罗（Robert DeNiro）的身材。 但是他的 7 部电影的平均收入却高达每部电影近 4.52 亿美元。

这些电影是什么？ 你可能知道《星球大战：C-3PO》中的 Droid C-3PO，那是金属机甲里面的安东尼·丹尼尔斯。 他扮演 C-3PO。



丹尼尔斯先生的全部电影（除了客串）都是由高收入的“星球大战”系列电影组成的。 这就解释了他的高平均收入和低电影数量。

类型和制作预算等变量，会影响电影数量与每部电影的平均收入之间的关联。 这个例子提醒人们，研究两个变量之间的关联，往往也涉及到了解其他相关的变量。

线形图

线形图是最常见的可视化图形之一，通常用于研究时序型的趋势和模式。

`movies_by_year` 表包含了 1980 年到 2015 年间，美国电影公司制作的电影的数据。这些列是：

列	内容
Year	年份
Total Gross	所有发行电影的国内总票房收入（以百万美元为单位）
Number of Movies	发行的电影数量
#1 Movie	收入最高的电影

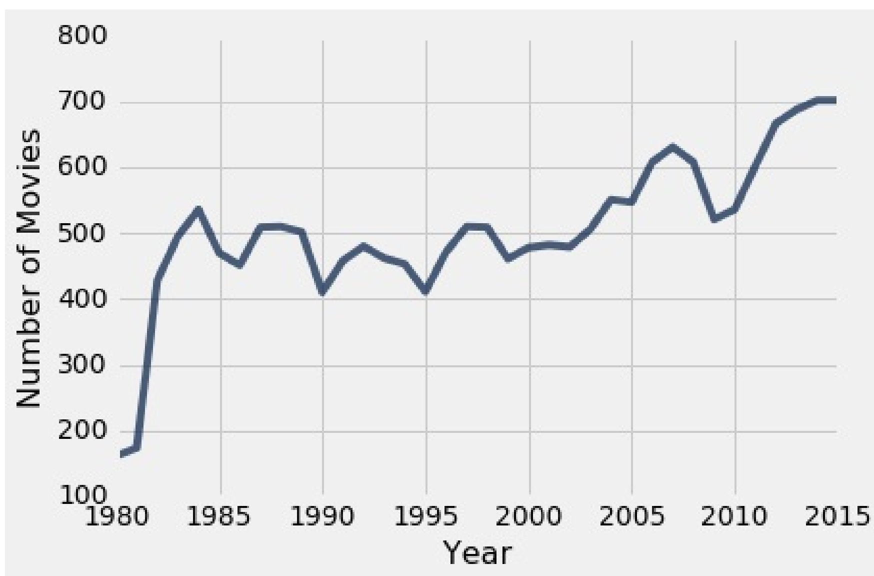
```
movies_by_year = Table.read_table('movies_by_year.csv')
movies_by_year
```

Year	Total Gross	Number of Movies	#1 Movie
2015	11128.5	702	Star Wars: The Force Awakens
2014	10360.8	702	American Sniper
2013	10923.6	688	Catching Fire
2012	10837.4	667	The Avengers
2011	10174.3	602	Harry Potter / Deathly Hallows (P2)
2010	10565.6	536	Toy Story 3
2009	10595.5	521	Avatar
2008	9630.7	608	The Dark Knight
2007	9663.8	631	Spider-Man 3
2006	9209.5	608	Dead Man's Chest

（省略了 26 行）

`Table` 的 `plot` 方法产生线形图。 它的两个参数与散点图相同：首先是横轴上的列，然后是纵轴上的列。 这是 1980 年到 2015 年间每年发行的电影数量的线形图。

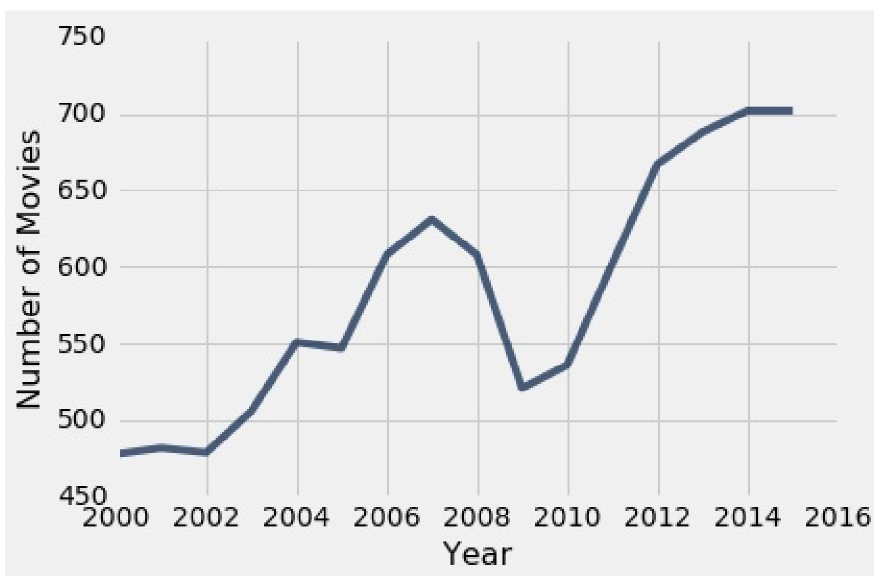
```
movies_by_year.plot('Year', 'Number of Movies')
```



虽然每年的数字都有明显的变化，但图形急剧上升，然后呈现平缓的上升趋势。 20 世纪 80 年代早期的剧增，部分是由于在上世纪 70 年代，电影制作人推动电影业的几年后，电影制片厂重新回到电影制作的前沿。

我们的重点将放在最近几年。 根据电影的主题，对应 2000 年到 2015 年的行，分配给名称 `century_21` 。

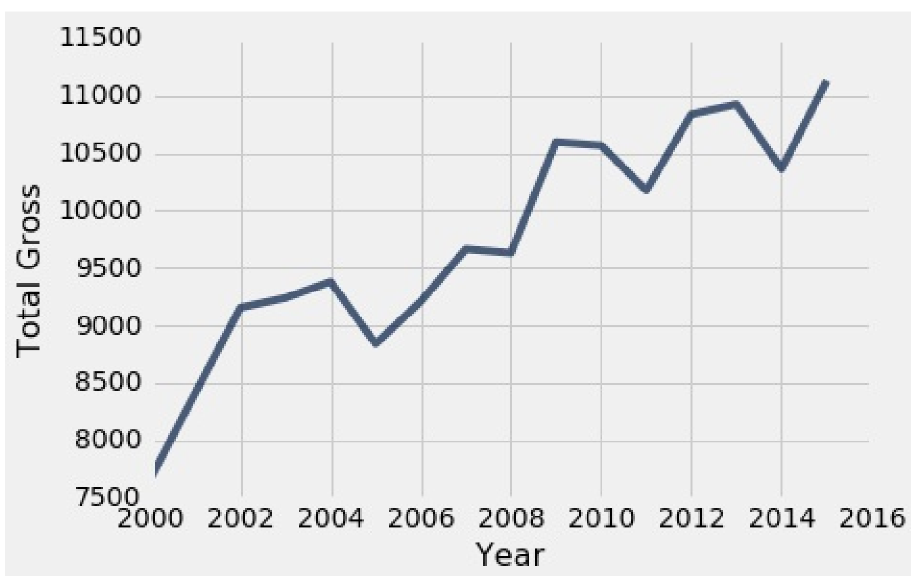
```
century_21 = movies_by_year.where('Year', are.above(1999))
century_21.plot('Year', 'Number of Movies')
```



2008 年的全球金融危机有明显的效果 - 2009 年发行的电影数量急剧下降。

但是，美元数量并没有太大的变化。

```
century_21.plot('Year', 'Total Gross')
```



尽管发生了金融危机，电影发行的数量也少得多，但 2009 年的国内总收入仍高于 2008 年。

造成这种矛盾的一个原因是，人们在经济衰退时往往会去看电影。“经济低迷时期，美国人涌向电影”，“纽约时报”于 2009 年 2 月说。文章引用南加州大学的马丁·卡普兰（Martin Kaplan）的话说：“人们想要忘记自己的烦恼，想和别人在一起。” 当节假日和昂贵的款待难以负担，电影提供了受欢迎的娱乐和宽慰。

2009 年的高票房收入的另一个原因是，电影《阿凡达》及其 3D 版本。 阿凡达不仅是 2009 年的第一部电影，它也是有史以来第二高的总票房电影，我们将在后面看到。

```
century_21.where('Year', are.equal_to(2009))
```

Year	Total Gross	Number of Movies	#1 Movie
2009	10595.5	521	Avatar

类别分许

可视化类别分布

许多数据不以数字的形式出现。 数据可以是音乐片段，或地图上的地方。 他们也可以是类别，你可以在里面放置个体。 以下是一些类别变量的例子。

- 个体是冰淇淋纸盒，变量就是纸盒里的味道。
- 个体是职业篮球运动员，变量是球员的队伍。
- 个体是年，而变量是今年最高票房电影的流派。
- 个体是调查对象，变量是他们从“完全不满意”，“有点满意”和“非常满意”中选择的回答。

`icecream` 表包含 30 盒冰激凌的数据。

```
icecream = Table().with_columns(
    'Flavor', make_array('Chocolate', 'Strawberry', 'Vanilla'),
    'Number of Cartons', make_array(16, 5, 9)
)
icecream
```

Flavor	Number of Cartons
Chocolate	16
Strawberry	5
Vanilla	9

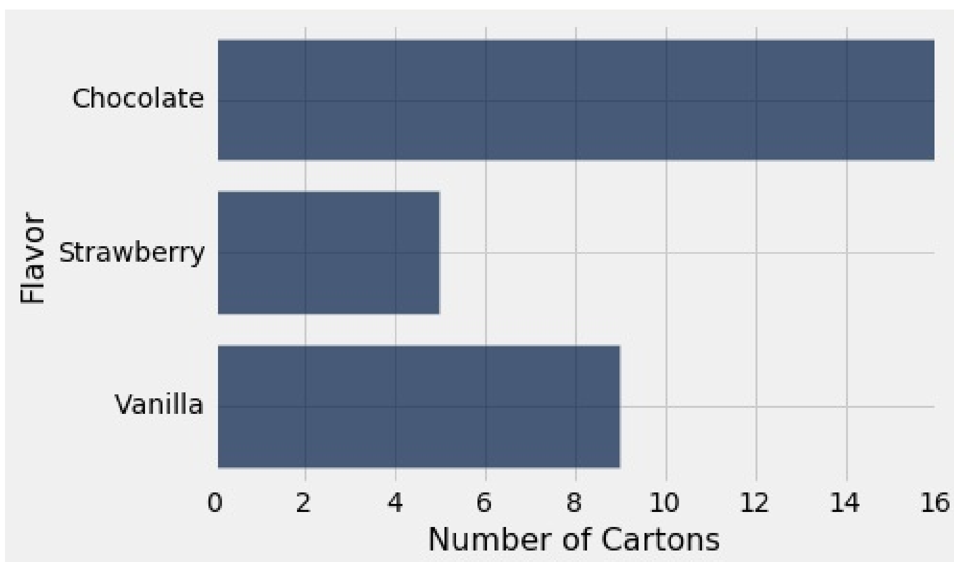
分类变量“口味”的值是巧克力，草莓和香草。 表格显示了每种口味的纸盒数量。 我们称之为分布表。 分布显示了所有变量的值，以及每个变量的频率。

条形图

条形图是可视化类别分布的熟悉方式。 它为每个类别显示一个条形。 条形的间隔相等，宽度相同。 每个条形的长度与相应类别的频率成正比。

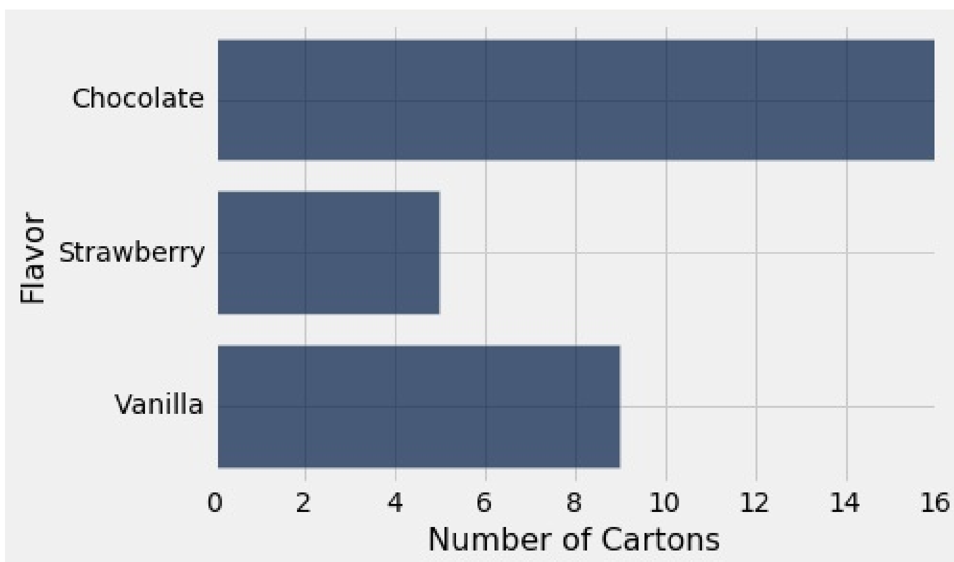
我们使用横条绘制条形图，因为这样更容易标注条形图。 所以 `Table` 的方法称为 `barh`。 它有两个参数：第一个是类别的列标签，第二个是频率的列标签。

```
icecream.barh('Flavor', 'Number of Cartons')
```



如果表格只包含一列类别和一列频率（如冰淇淋），则方法调用甚至更简单。 你可以指定包含类别的列， `barh` 将使用另一列中的值作为频率。

```
icecream.barh('Flavor')
```



类别分布的特征

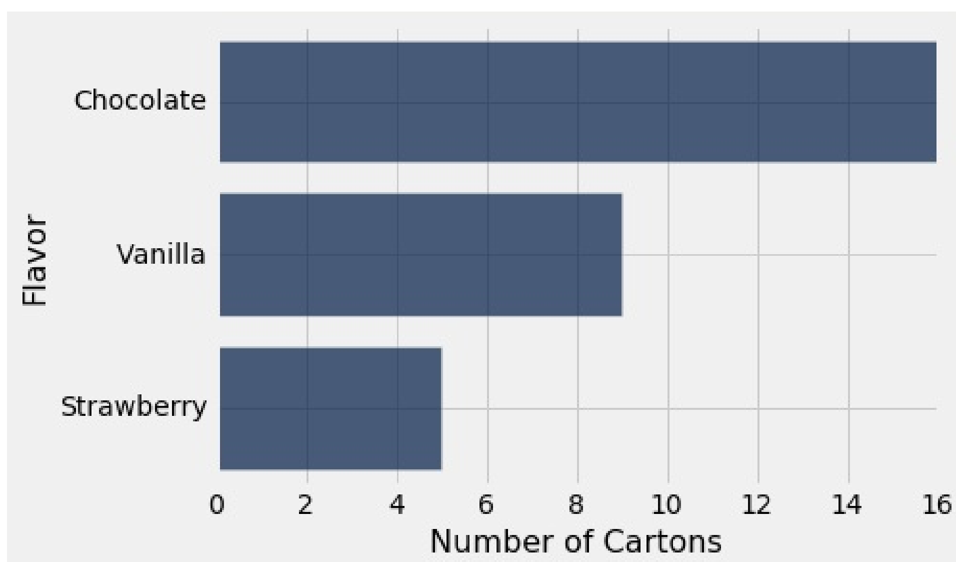
除了纯粹的视觉差异之外，条形图和我们在前面章节中看到的两个图表之间还有一个重要的区别。 它们是散点图和线图，两者都显示两个数值变量 - 两个轴上的变量都是数值型的。 相比之下，条形图的一个轴上是类别，在另一个轴上具有数值型频率。

这对图表有影响。首先，每个条形的宽度和相邻条形之间的间隔完全取决于生成图的人，或者用于生成该图的程序。 Python 为我们做了这些选择。 如果你要手动绘制条形图，则可以做出完全不同的选择，并且仍然会是完全正确的条形图，前提是你使用相同宽度绘制了所有条形，并使所有间隔保持相同。

最重要的是，条形可以以任何顺序绘制。“巧克力”，“香草”和“草莓”这些类别没有普遍的等级顺序，不像数字 `5, 7` 和 `10` 。

这意味着我们可以绘制一个易于解释的条形图，方法是按降序重新排列条形图。 为了实现它，我们首先按照 `Number of Cartons` 的降序，重新排列 `icecream` 的行，然后绘制条形图。

```
icecream.sort('Number of Cartons', descending=True).barh('Flavor')
```



这个条形图包含的信息和以前的完全一样，但是它更容易阅读。 虽然在只读三个条形的情况下，这并不是一个巨大的收益，但是当分类数量很大时，这可能是相当重要的。

组合分类数据

为了构造 `icecream` 表，有人不得不查看 30 个冰淇淋盒子，并计算每种口味的数量。 但是，如果我们的数据还没有包含频率，我们必须在绘制条形图之前计算频率。 这是一个例子，其中它是必要的。

`top` 表由美国历史上最畅销的电影组成。 第一列包含电影的标题；《星球大战：原力觉醒》（Star Wars: The Force Awakens）排名第一，美国票房总额超过 9 亿美元。 第二列包含制作电影的工作室的名称。 第三列包含国内票房收入（美元），第四列包含按 2016 年价格计算的，票面总收入。 第五列包含电影的发行年份。

列表中有 200 部电影。 根据未调整的总收入，这是前十名。

```
top = Table.read_table('top_movies.csv')
top
```

Title	Studio	Gross	Gross (Adjusted)	Year
Star Wars: The Force Awakens	Buena Vista (Disney)	906723418	906723400	2015
Avatar	Fox	760507625	846120800	2009
Titanic	Paramount	658672302	1178627900	1997
Jurassic World	Universal	652270625	687728000	2015
Marvel's The Avengers	Buena Vista (Disney)	623357910	668866600	2012
The Dark Knight	Warner Bros.	534858444	647761600	2008
Star Wars: Episode I - The Phantom Menace	Fox	474544677	785715000	1999
Star Wars	Fox	460998007	1549640500	1977
Avengers: Age of Ultron	Buena Vista (Disney)	459005868	465684200	2015
The Dark Knight Rises	Warner Bros.	448139099	500961700	2012

（省略了 190 行）

迪斯尼的子公司布埃纳维斯塔（Buena Vista）就像福克斯（Fox）和华纳兄弟（Warner Brothers）一样，经常出现在前十名中 如果我们从 200 行中看，哪个工作室最常出现？

为了解决这个问题，首先要注意的是，我们需要的只是一个拥有电影和工作室的表格；其他信息是不必要的。

```
movies_and_studios = top.select('Title', 'Studio')
```

`Table` 的 `group` 方法组允许我们，通过将每个工作室当做一个类别，并将每一行分配给一个类别，来计算每个工作室出现在表中的频率。 `group` 方法将包含类别的列标签作为其参数，并返回每个类别中行数量的表格。 数量列始终称为 `count` ，但如果你希望的话，则可以使用 `relabeled` 更改该列。

```
movies_and_studios.group('Studio')
```

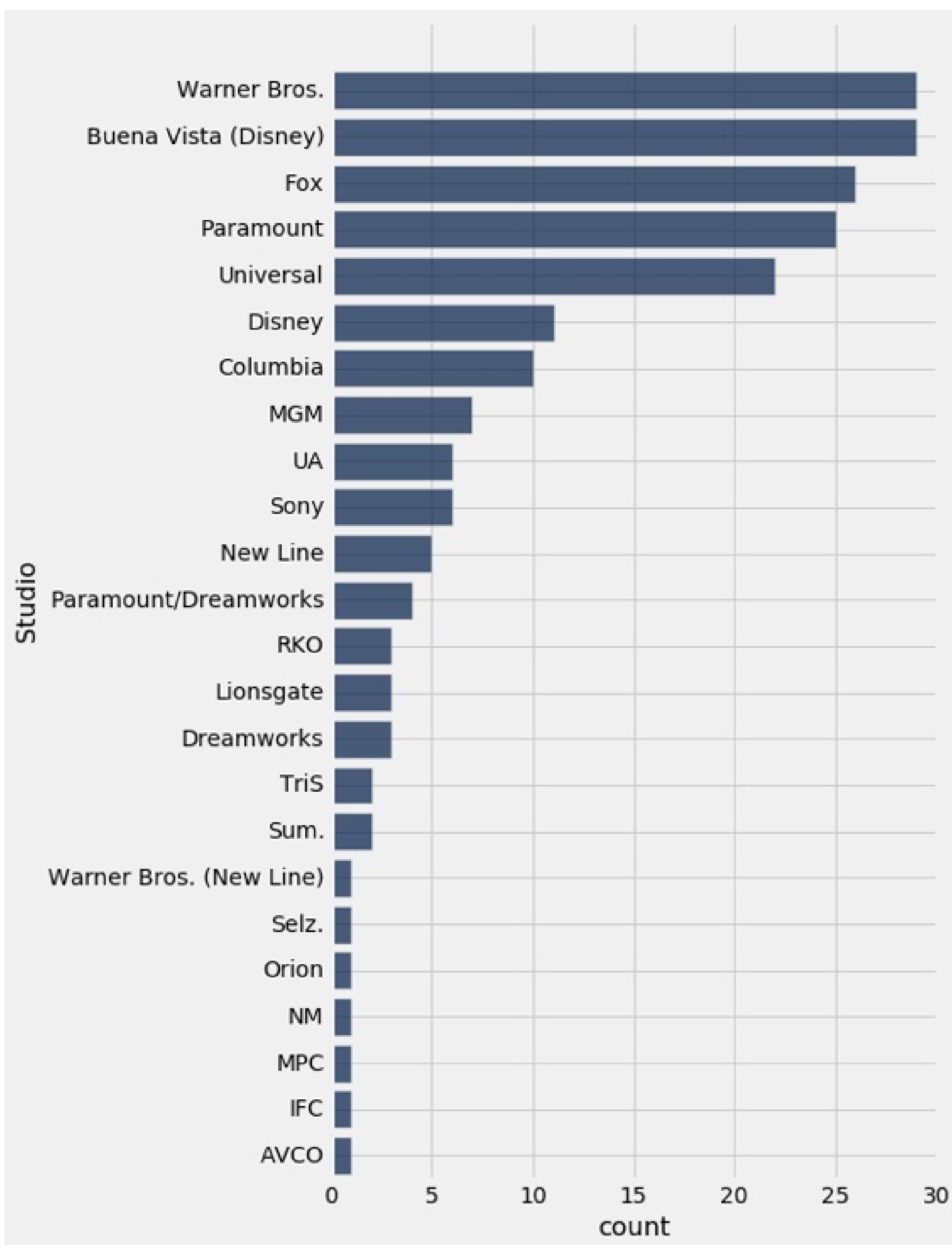
Studio	count
AVCO	1
Buena Vista (Disney)	29
Columbia	10
Disney	11
Dreamworks	3
Fox	26
IFC	1
Lionsgate	3
MGM	7
MPC	1

（省略了 14 行）

因此，`group` 创建一个分布表，显示电影在类别（工作室）之间如何分布。

现在我们可以使用这个表格，以及我们上面获得的图形技能来绘制条形图，显示前 200 个最高收入的电影中，哪个工作室是最常见的。

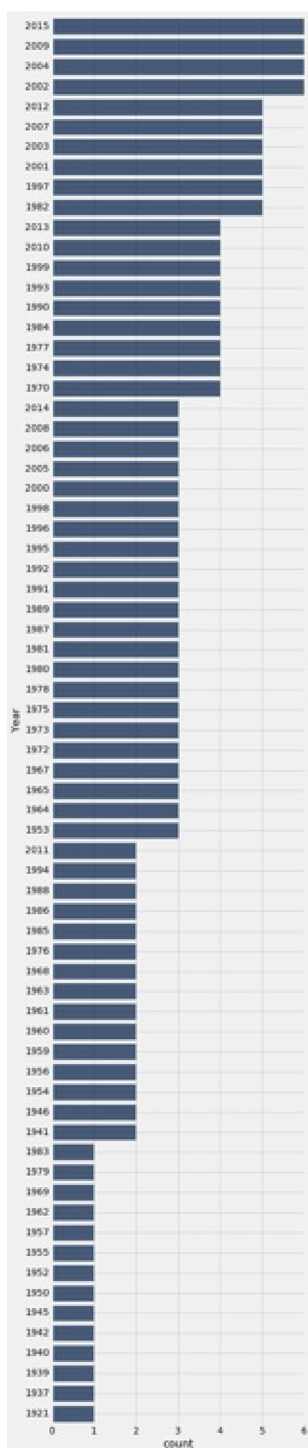
```
studio_distribution = movies_and_studios.group('Studio')
studio_distribution.sort('count', descending=True).barh('Studio')
```



华纳兄弟（Warner Brothers）和布埃纳维斯塔（Buena Vista）是前 200 电影中最常见的工作室。 华纳兄弟制作了哈利波特电影，布埃纳维斯塔制作了星球大战。

由于总收入以未经调整的美元来衡量，所以最近几年的顶级电影比过去几十年更频繁，这并不令人惊讶。 以绝对数量来看，现在的电影票价比以前更高，因此总收入也更高。 这是通过条形图证明的，这些条形图显示了 200 部电影的发行年份。

```
movies_and_years = top.select('Title', 'Year')
movies_and_years.group('Year').sort('count', descending=True).barh('Year')
```



所有最长的条形都对应 2000 年以后的年份。这与我们的观察一致，即最近几年应该是最频繁的。

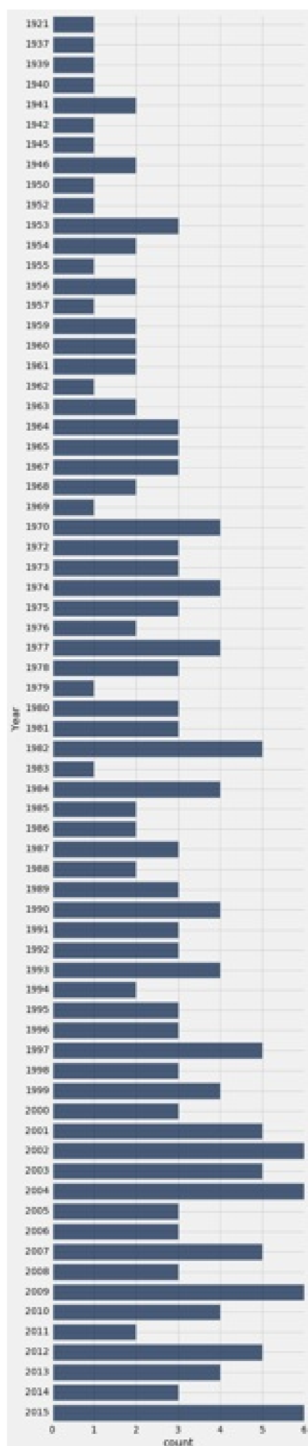
面向数值变量

这张图有一些未解决的地方。 虽然它确实回答了这个问题，200 部最受欢迎的电影中，最常见的发行年份，但并没有按时间顺序列出所有年份。 它将年作为一个分类变量。

但是，年份是固定的时序单位，确实拥有顺序。 他们也有相对于彼此的固定的数值距离。 让我们看看当我们试图考虑它的时候会发生什么。

默认情况下，`barh` 将类别（年）从最低到最高排序。 所以我们将运行这个代码，但不按 `count` 进行排序。

```
movies_and_years.group('Year').barh('Year')
```



现在年份是升序了。 但是这个条形图还是有点问题。 1921 年和 1937 年的条形与 1937 年和 1939 年的条形相距甚远。条形图并没有显示出，200 部电影中没有一部是在 1922 年到 1936 年间发布的。基于这种可视化，这种不一致和遗漏，使早期年份的分布难以理解。

条形图用做类别变量的可视化。 当变量是数值，并且我们创建可视化时，必须考虑其值之间的数值关系。 这是下一节的主题。

数值分布

可视化数值分布

数据科学家研究的许多变量是定量的或数值的。它们的值是你可以做算术的数字。我们所看到的例子包括一本书的章节数量，电影的收入以及美国人的年龄。

类别变量的值可以按照数字编码，但是这不会使变量成为定量的。在我们研究的，按年龄组分类的人口普查数据的例子中，分类变量 `SEX` 中，`'Male'` 的数字代码为 `1`，`'Female'` 的数字代码为 `2`，以及分组 `1` 和 `2` 的合计为 `0`。 `1` 和 `2` 是数字，在这种情况下，从 `2` 中减 `1` 或取 `0,1` 和 `2` 的平均值，或对这三个值执行其他算术是没有意义的。 `SEX` 是一个类别变量，即使这些值已经赋予一个数字代码。

对于我们的主要示例，我们将返回到我们在可视化分类数据时，所研究的数据集。这是一个表格，它由美国历史上最畅销的电影中的数据组成。为了方便起见，这里再次描述表格。

第一列包含电影的标题。第二列包含制作电影的工作室的名称。第三个包含国内票房总值（美元），第四个包含按 2016 年价格计算的票面收入总额。第五个包含电影的发行年份。

列表中有 200 部电影。根据 `Gross` 列中未调整的总收入，这是前十名。

```
top = Table.read_table('top_movies.csv')
# Make the numbers in the Gross and Gross (Adjusted) columns look nicer:
top.set_format([2, 3], NumberFormatter)
```

Title	Studio	Gross	Gross (Adjusted)	Year
Star Wars: The Force Awakens	Buena Vista (Disney)	906,723,418	906,723,400	2015
Avatar	Fox	760,507,625	846,120,800	2009
Titanic	Paramount	658,672,302	1,178,627,900	1997
Jurassic World	Universal	652,270,625	687,728,000	2015
Marvel's The Avengers	Buena Vista (Disney)	623,357,910	668,866,600	2012
The Dark Knight	Warner Bros.	534,858,444	647,761,600	2008
Star Wars: Episode I - The Phantom Menace	Fox	474,544,677	785,715,000	1999
Star Wars	Fox	460,998,007	1,549,640,500	1977
Avengers: Age of Ultron	Buena Vista (Disney)	459,005,868	465,684,200	2015
The Dark Knight Rises	Warner Bros.	448,139,099	500,961,700	2012

（省略了 190 行）

可视化调整后收入的分布

在本节中，我们将绘制 `Gross (Adjusted)` 列中数值变量的分布图。 为了简单起见，我们创建一个包含我们所需信息的小表。 而且由于三位数字比九位数字更容易处理，我们以百万美元衡量调整后的总收入。 注意如何使用舍入仅保留两位小数。

```
millions = top.select(0).with_column('Adjusted Gross',
                                     np.round(top.column(3)/1e6, 2))
millions
```

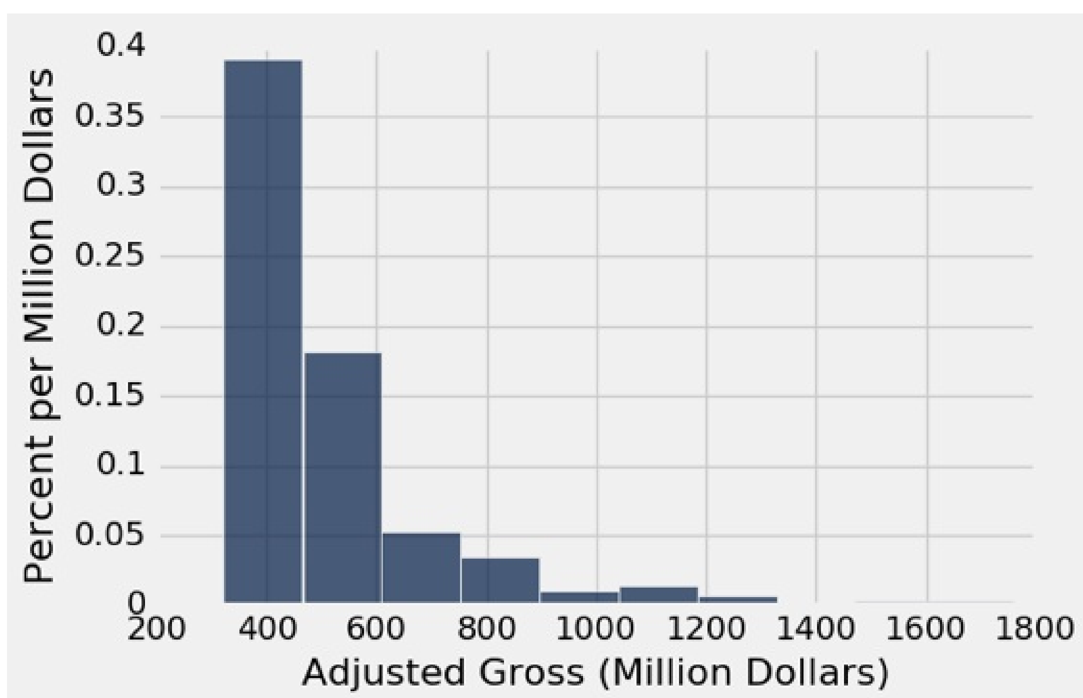
Title	Adjusted Gross
Star Wars: The Force Awakens	906.72
Avatar	846.12
Titanic	1178.63
Jurassic World	687.73
Marvel's The Avengers	668.87
The Dark Knight	647.76
Star Wars: Episode I - The Phantom Menace	785.72
Star Wars	1549.64
Avengers: Age of Ultron	465.68
The Dark Knight Rises	500.96

直方图

数值数据集的直方图看起来非常像条形图，虽然它有一些重要的差异，我们将在本节中讨论。 首先，我们只画出调整后收入的直方图。

hist方法生成列中值的直方图。 可选的单位参数用于两个轴上的标签。 直方图显示调整后的总额分布，以百万美元为单位。

```
millions.hist('Adjusted Gross', unit="Million Dollars")
```



横轴

这些金额已被分组为连续的间隔，称为桶。尽管在这个数据集中，没有电影正好在两个桶之间的边缘上，但是 `hist` 必须考虑数值可能在边缘的情况。所以 `hist` 有一个端点约定：`bin` 包含左端点的数据，但不包含右端点的数据。

我们使用符号 `[a, b)` 表示从 `a` 开始并在 `b` 结束但不包括 `b` 的桶。

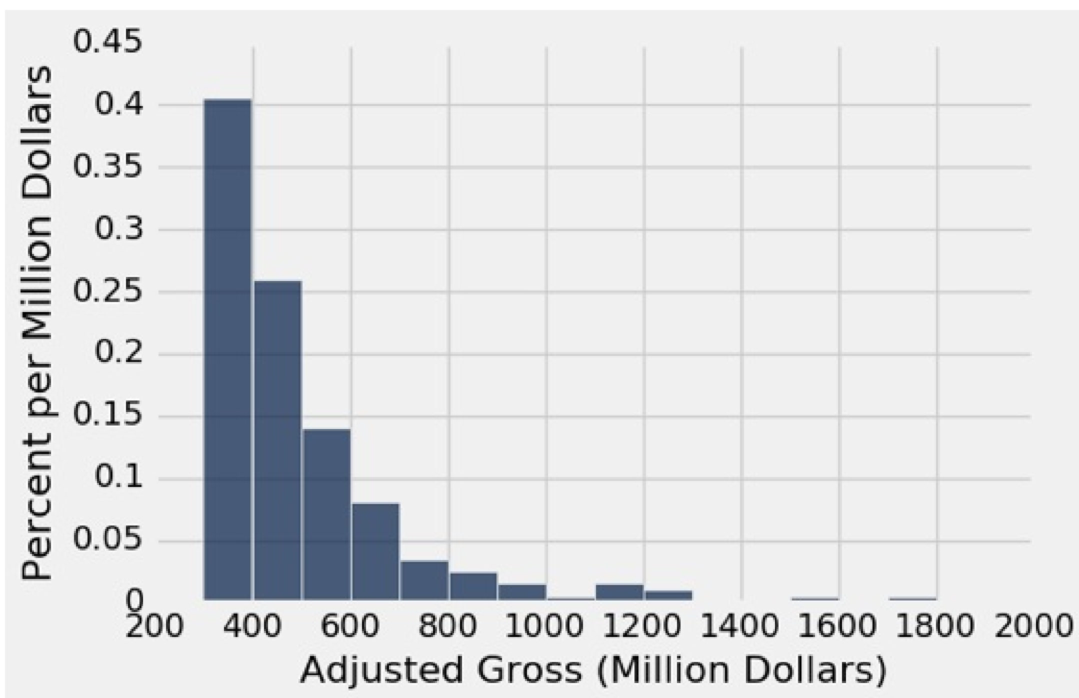
有时，必须在第一个或最后一个箱中进行调整，以确保包含变量的最小值和最大值。在前面研究的人口普查数据中，你看到了一个这样的调整的例子，其中“100”岁的年龄实际上意味着“100 岁以上”。

我们可以看到，有 10 个桶（有些桶很低，难以看到），而且它们的宽度都是一样的。我们也可以看到，没有一部电影的收入不到三亿美元，那是因为我们只考虑有史以来最畅销的电影。

准确看到桶的末端在哪里，有点困难。例如，精确地确定值 500 位于横轴上的位置并不容易。所以很难判断一个条形的结束位置和下一个条形的开始位置。

可选参数 `bins` 可以与 `hist` 一起使用来指定桶的端点。它必须由一系列数字组成，这些数字以第一个桶的左端开始，以最后一个桶的右端结束。我们首先将桶中的数字设置为 `300,400,500` 等等，以 `2000` 结尾。

```
millions.hist('Adjusted Gross', bins=np.arange(300,2001,100), unit="Million Dollars")
```



这个图的横轴比较容易阅读。 标签 `200,400,600` 等以对应的值为中心。 最高的条形是对应三亿到四亿美元之间的电影。

少数电影投入了 8 亿美元甚至更多。 这导致这个数字“向右倾斜”，或者更不正式地说，“右侧长尾”。 大量人口的收入或租金等变量的分布也经常具有这种形式。

桶的数量

可以使用`bin`方法从一个表格中计算出桶中的值的数量，该方法接受列标签或索引，以及可选的序列或桶的数量。 结果是直方图的表格形式。 第一列列出了桶的左端点（但请参阅下面关于最终值的注释）。 第二列包含`Adjusted Gross`列中所有值在相应桶中的数量。 也就是说，它计数所有`Adjusted Gross`的所有值，它们大于或等于`bin`中的值，但小于下一个`bin`中的值。

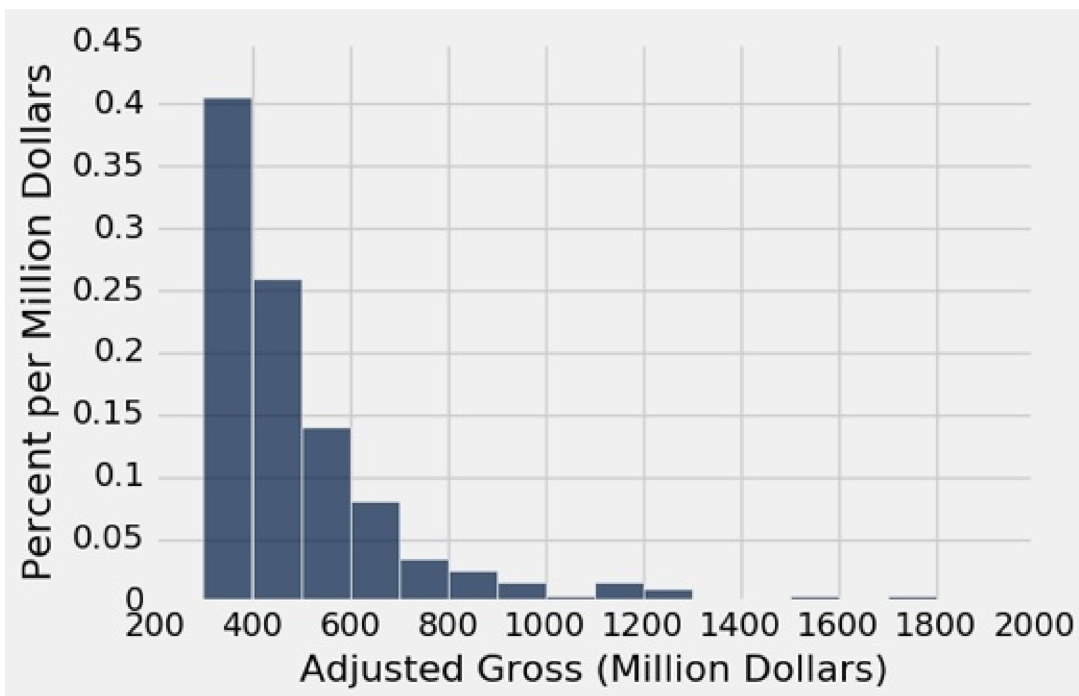
```
bin_counts = millions.bin('Adjusted Gross', bins=np.arange(300,2001,100))
bin_counts.show()
```

bin	Adjusted Gross count
300	81
400	52
500	28
600	16
700	7
800	5
900	3
1000	1
1100	3
1200	2
1300	0
1400	0
1500	1
1600	0
1700	1
1800	0
1900	0
2000	0

注意最后一行的 `bin` 值 2000。 这不是任何条形的左端点 - 这是最后一个条形的右端点。 按照端点约定，那里的数据不包括在内。 因此，相应的计数记录为 0，并且即使已经有超过二十亿美元的电影也被记录为 0。 当 `bin` 或 `hist` 使用 `bin` 参数调用时，图只考虑在指定 `bin` 中的值。

一旦数值已经分入桶中，所得数量可以用来使用 `bin_column` 命名参数来生成直方图，以指定哪个列包含桶的下界。

```
bin_counts.hist('Adjusted Gross count', bin_column='bin', unit='Million Dollars')
```



纵轴：密度刻度

一旦我们已经照顾到细节，如桶的末端，直方图的横轴易于阅读。 纵轴的特征需要更多关注。 我们会一一讲解。

我们先来看看如何计算垂直轴上的数字。 如果计算看起来有些奇怪，请耐心等待 - 本节的其余部分将解释原因。

计算。每个条形的高度是桶中的元素的百分比，除以桶的宽度。

> 译者注：存在很多种直方图，比如频数直方图、频率质量直方图和频率密度直方图。它们的纵轴数值不相同，但是图形形状是一样的。这里是最后一种，频率密度直方图。

```
counts = bin_counts.relabeled('Adjusted Gross count', 'Count')
percents = counts.with_column(
    'Percent', (counts.column('Count')/200)*100
    )
heights = percents.with_column(
    'Height', percents.column('Percent')/100
    )
heights
```

bin	Count	Percent	Height
300	81	40.5	0.405
400	52	26	0.26
500	28	14	0.14
600	16	8	0.08
700	7	3.5	0.035
800	5	2.5	0.025
900	3	1.5	0.015
1000	1	0.5	0.005
1100	3	1.5	0.015
1200	2	1	0.01

（省略了 8 行）

在上面直方图的纵轴上查看数字，检查列高度是否正确。

如果我们只查看表格的第一行，计算就会变得清晰。

请记住，数据集中有 200 部电影。这个 `[300,400)` 的桶包含 81 部电影。这是所有电影的 40.5%：$\text{Percent} = \frac{81}{200} \cdot 100 = 40.5$。

`[300, 400)` 桶的宽度是 `400-300 = 100`。所以 $\text{Height} = \frac{40.5}{100} = 0.405$。

用于计算高度的代码使用了总共有 200 个电影，以及每个箱的宽度是 100 的事实。

单位。条形的高度是 40.5% 除以 1 亿美元，因此高度是 0.405% 每百万美元。

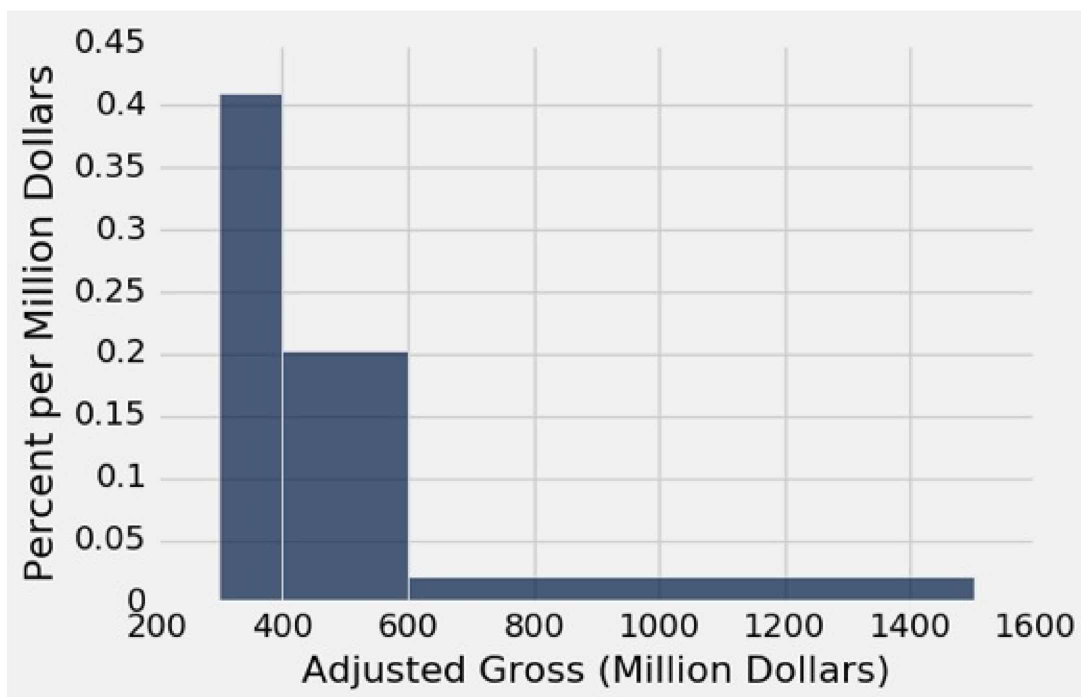
这种绘制直方图的方法创建了一个垂直轴，它是在密度刻度上的。条形的高度不是桶中条目的百分比；它是桶中的条目除以桶的宽度。这就是为什么高度衡量拥挤度或密度。

让我们看看为什么这很重要。

不等的桶

直方图相比条形图的一个优点是，直方图可以包含不等宽度的桶。 以下将 `Millions` 中的值分为三个不均匀的类别。

```
uneven = make_array(300, 400, 600, 1500)
millions.hist('Adjusted Gross', bins=uneven, unit="Million Dollars")
```



这里是三个桶中的数量。

```
millions.bin('Adjusted Gross', bins=uneven)
```

bin	Adjusted Gross count
300	81
400	80
600	37
1500	0

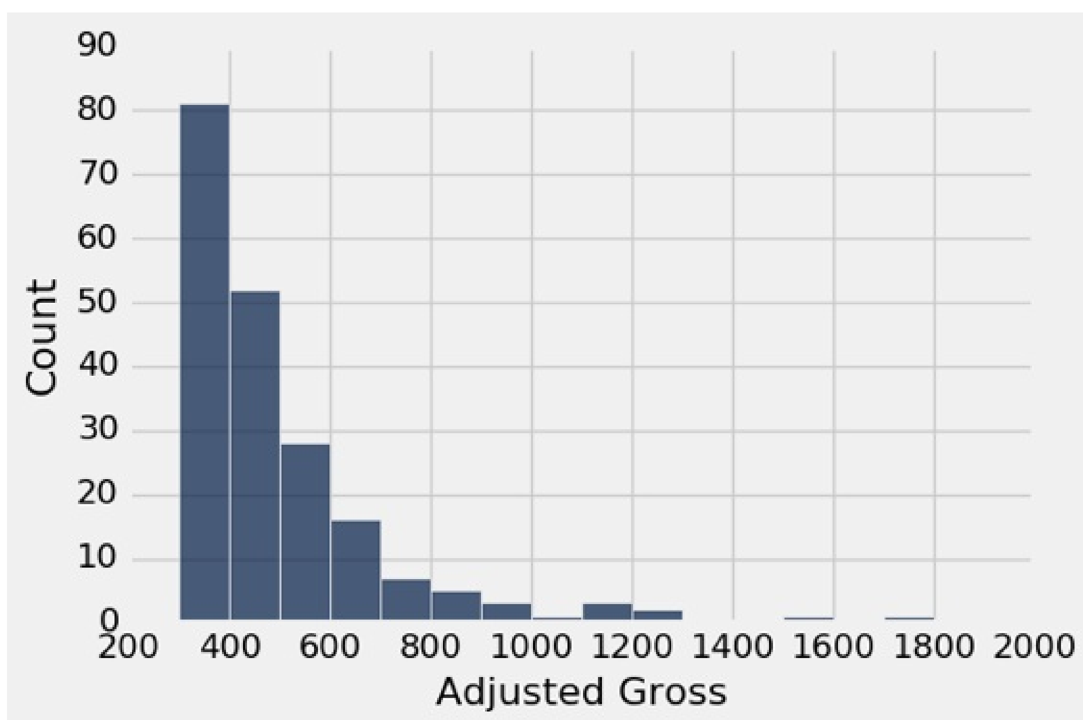
虽然范围 `[300,400)` 和 `[400,600)` 具有几乎相同的计数，但前者的高度是后者的两倍，因为它只有一半的宽度。 `[300,400)` 中的值的密度是 `[400,600)` 中的密度的两倍。

直方图帮助我们可视化数轴上数据最集中的地方，特别是当桶不均匀的时候。

仅仅绘制数量的问题

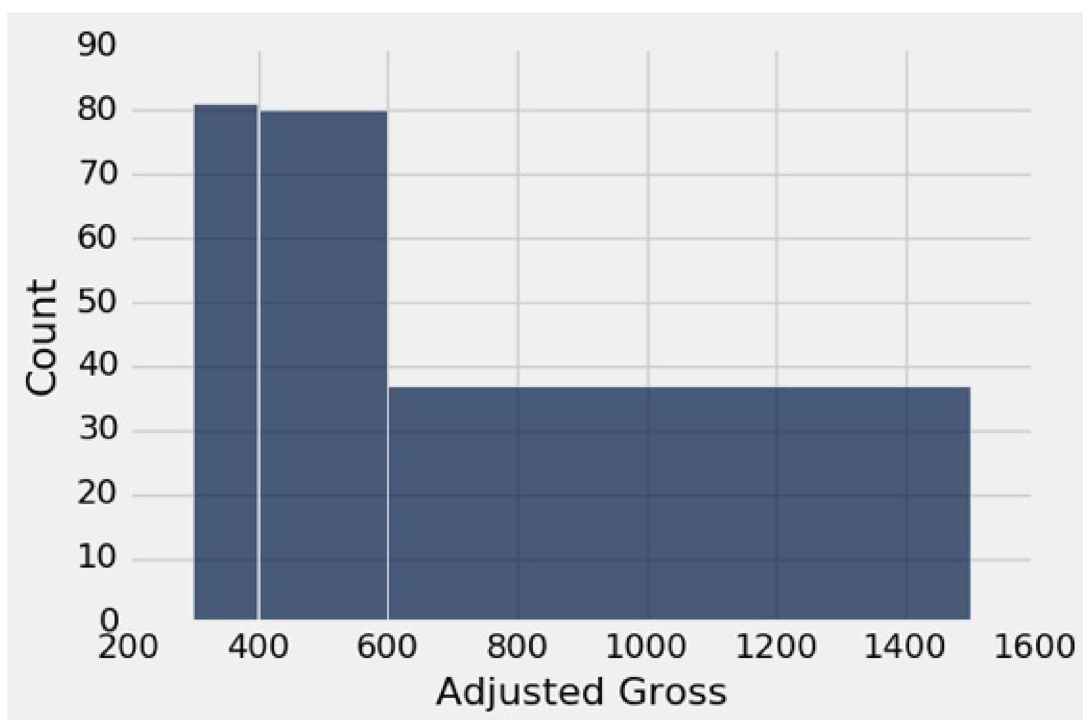
可以使用 `hist` 方法的 `normed=False` 选项直接在图表中显示数量。 生成的图表与直方图具有相同的形状，但这些桶的宽度均相等，尽管纵轴上的数字不同。

```
millions.hist('Adjusted Gross', bins=np.arange(300,2001,100), normed=False)
```



虽然数量刻度可能比密度刻度更自然，但当桶宽度不同时，图表高度的有误导性。 下面看起来（由于计数）高收入电影相当普遍，事实上我们已经看到它们相对较少。

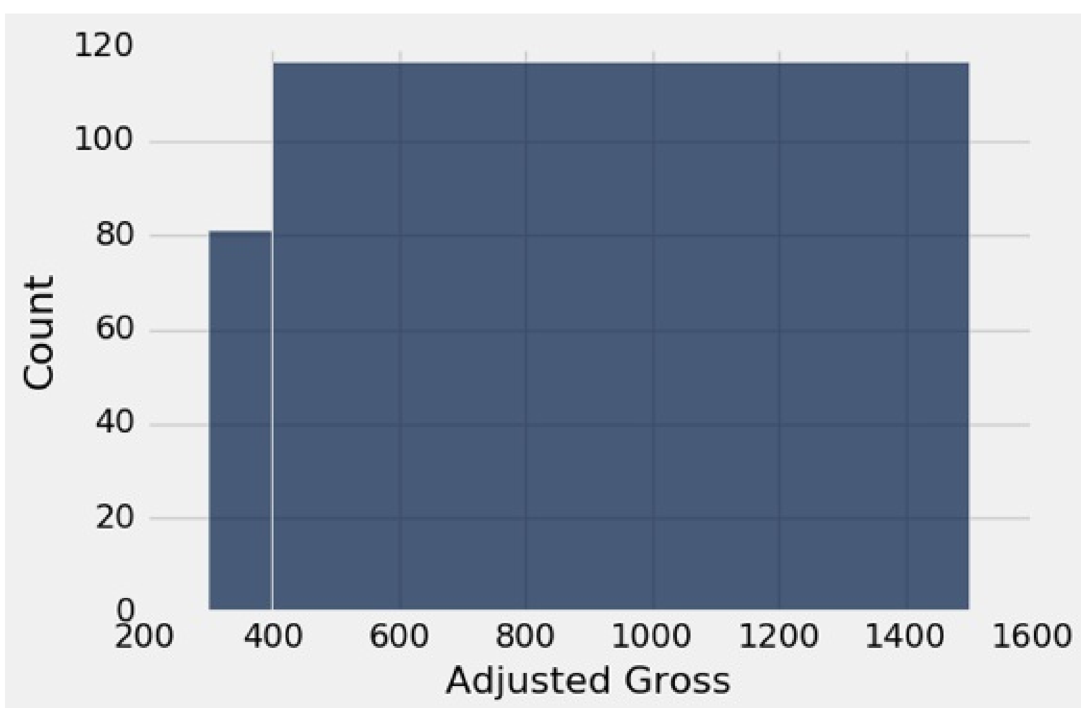
```
millions.hist('Adjusted Gross', bins=uneven, normed=False)
```



即使使用的方法被称为 `hist`，上面的图不是一个直方图。 误导性地夸大了至少 6 亿美元的电影比例。 每个桶的高度只是按照桶中的电影数量绘制，而不考虑桶宽度的差异。

如果最后两个桶组合起来，情况就变得更加荒谬了。

```
very_uneven = make_array(300, 400, 1500)
millions.hist('Adjusted Gross', bins=very_uneven, normed=False)
```



在这个基于数量的图像中，电影分布完全失去了形状。

直方图：通用原则和计算

上图显示，眼睛将面积视为“较大”的东西，而不是高度。当桶的宽度不同时，这种观察变得尤为重要。

这就是直方图具有两个定义属性的原因：

- 桶按比例绘制并且是连续的（尽管有些可能是空的），因为横轴上的值是数值型的。
- 每个条形的面积与桶中的条目数成比例。

属性（2）是绘制直方图的关键，通常实现如下：

```
条形的面积 = 桶中条目的百分比
```

高度的计算仅仅使用了一个事实，条形是长方形的。

```
条形的面积 = 条形的高度 * 桶的宽度
```

因此，

```
条形的高度 = 条形的面积 / 桶的宽度
          = 桶中条目的百分比 / 桶的宽度
```

高度的单位是“百分比每横轴单位”。

当使用这种方法绘制时，直方图被称为在密度刻度上绘制。 在这个刻度上：

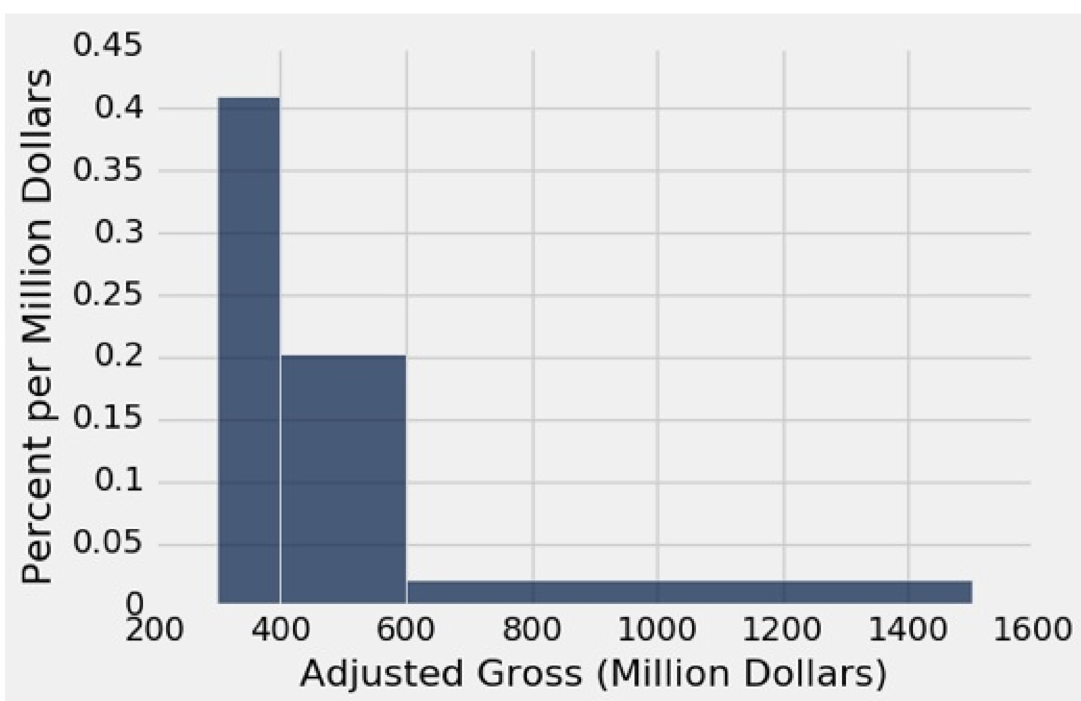
- 每个条形的面积等于相应桶中的数据值的百分比。
- 直方图中所有条形的总面积为 100%。 从比例的角度来讲，我们说直方图中所有条形的面积“总计为 1”。

平顶和细节水平

即使密度刻度使用面积正确表示了百分比，但是通过将值分组到桶中，丢失了一些细节。

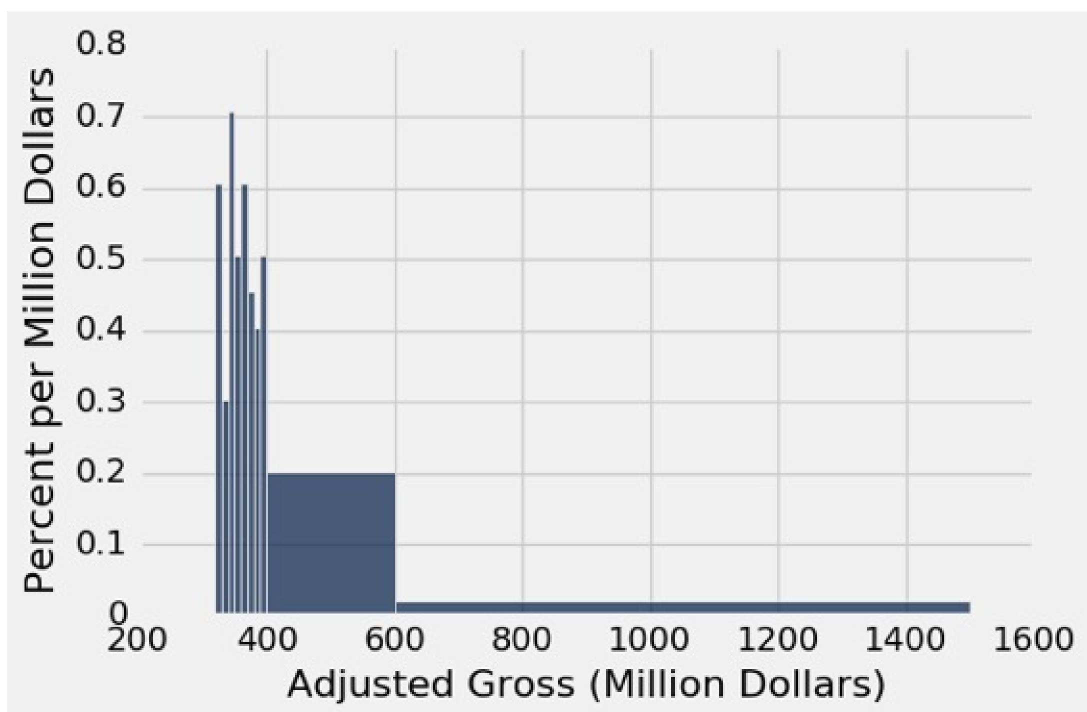
再看一下下图中的 `[300,400)` 的桶。 值为“0.405% 每百万美元”的桶的平顶，隐藏了电影在这个桶分布有些不均匀的事实。

```
millions.hist('Adjusted Gross', bins=uneven, unit="Million Dollars")
```



为了看到它，让我们将 `[300, 400)` 划分为更窄的 10 个桶。每个桶的宽度都是一千万美元。

```
some_tiny_bins = make_array(300, 310, 320, 330, 340, 350, 360, 370, 380, 390, 400, 600
, 1500)
millions.hist('Adjusted Gross', bins=some_tiny_bins, unit='Million Dollars')
```

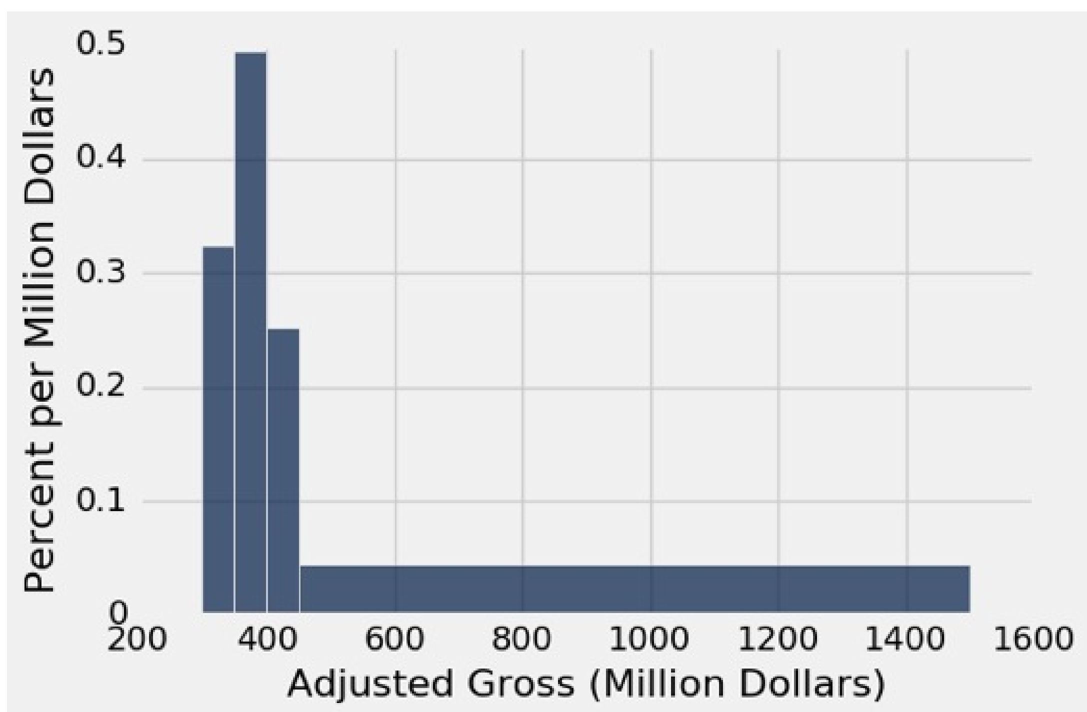


直方图 Q&A

让我们再画一遍直方图，这次只有四个桶，检查我们对概念的理解。

```
uneven_again = make_array(300, 350, 400, 450, 1500)
millions.hist('Adjusted Gross', bins=uneven_again, unit='Million Dollars')

millions.bin('Adjusted Gross', bins=uneven_again)
```



bin	Adjusted Gross count
300	32
350	49
400	25
450	92
1500	0

再次查看直方图，并将 `[400,450)` 的桶与 `[450,1500)` 桶进行比较。

问：哪个桶里面有更多的电影？

答： `[450,1500)` 的桶。它有 92 部电影，而 `[400,450)` 桶中有 25 部电影。

问：那么为什么 `[450,1500)` 的桶比 `[400,450)` 桶短得多呢？

答：因为高度代表桶里每单位空间的密度，而不是桶里的电影数量。 `[450,1500)` 的桶中的电影确实比 `[400,450)` 的桶多，但它也是一个大桶。 所以它不那么拥挤。 其中的电影密度要低得多。

条形图和直方图的区别

- 条形图为每个类别展示一个数量。 它们通常用于显示类别变量的分布。 直方图显示定量变量的分布。
- 条形图中的所有条形都具有相同的宽度，相邻的条形之间有相等的间距。 直方图的条形可以具有不同的宽度，并且是连续的。
- 条形图中条形的长度（或高度，如果垂直绘制）与每个类别的值成正比。 直方图中条形的高度是密度的度量；直方图中的条形的面积与桶中的条目数量成正比。

重叠的图表

在这一章中，我们学习了如何通过绘制图表来显示数据。 这种可视化的常见用法是比较两个数据集。 在本节中，我们将看到如何叠加绘图，即将它们绘制在单个图形中，拥有同一对坐标轴

为了使重叠有意义，重叠的图必须表示相同的变量并以相同的单位进行测量。

为了绘制重叠图，可以用相同的方法调用 `scatter` ， `plot` 和 `barh` 方法。 对于 `scatter` 和 `plot` ，一列必须作为所有叠加图的公共横轴。 对于 `barh` ，一列必须作为一组类别的公共轴。 一般的调用看起来像这样：

```
name_of_table.method(column_label_of_common_axis, array_of_labels_of_variables_to_plot
)
```

更常见的是，我们首先仅仅选取图表所需的列。之后通过指定共同轴上的变量来调用方法。

```
name_of_table.method(column_label_of_common_axis)
```

散点图

高尔顿（Franics Galton，1822 ~ 1911 年）是一位英国博学家，他是分析数值变量之间关系的先驱。 他对有争议的优生学领域特别感兴趣，实际上，他创造了这个术语 - 这涉及到如何将物理特征从一代传到下一代。

高尔顿精心收集了大量的数据，其中一些我们将在本课程中分析。 这是高尔顿的，有关父母及其子女身高的数据的子集。 具体来说，数据由 179 名男性组成，他们在家庭中第一个出生。数据是他们自己的高度和父母的高度。所有的高度都是以英寸来测量的。

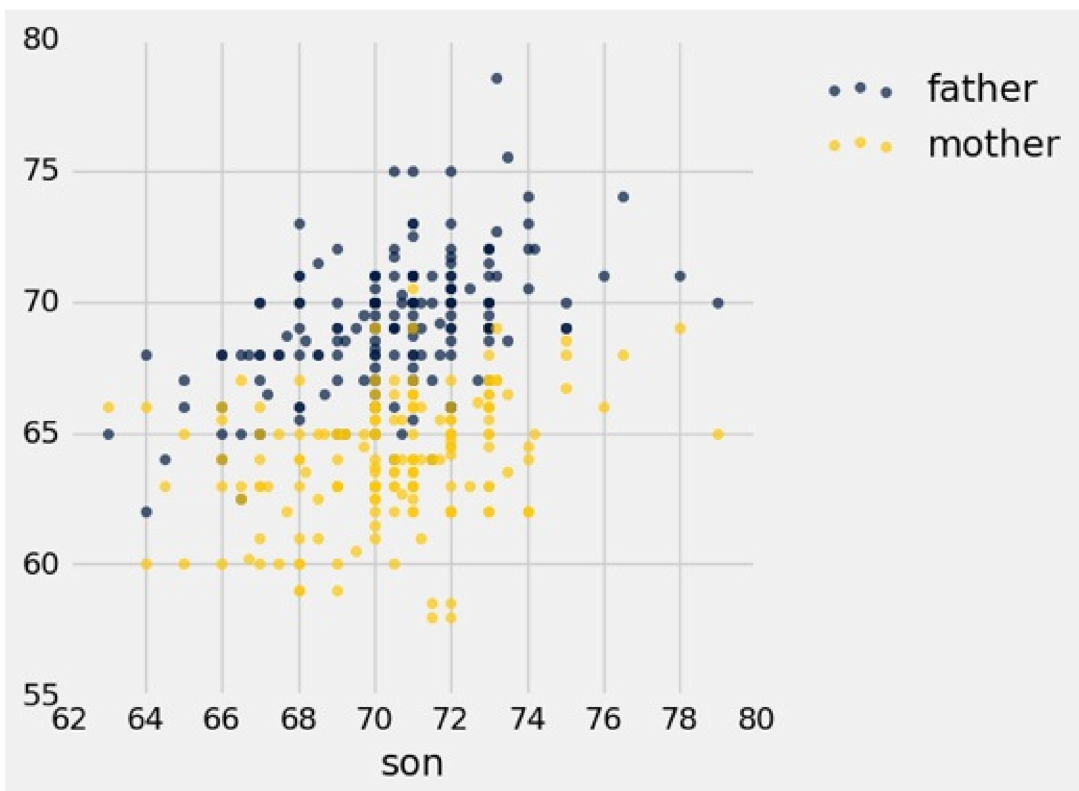
```
heights = Table.read_table('galton_subset.csv')
heights
```

father	mother	son
78.5	67	73.2
75.5	66.5	73.5
75	64	71
75	64	70.5
75	58.5	72
74	68	76.5
74	62	74
73	67	71
73	67	68
73	66.5	71

（省略了 169 行）

`scatter` 方法使我们能够可视化，儿子的身高如何与父母的身高有关。 在图中，儿子的身高将形成公共的横轴。

```
heights.scatter('son')
```



注意我们仅仅指定了公共的横轴上的变量（儿子的身高）。 Python 绘制了两个散点图：这个变量和另外两个之间的关系，每个关系一个。

金色和蓝色的散点图向上倾斜，并显示出儿子的高度和父母的高度之间的正相关。 蓝色（父亲）的绘图一般比金色高，因为父亲一般比母亲高。

线形图

我们的下一个例子涉及更近的儿童数据。 我们将返回到人口普查数据表 `us_pop` ，再次在下面创建用于参考。 由此，我们将提取 0 至 18 岁年龄段的所有儿童的数量。

```
# Read the full Census table
census_url = 'http://www2.census.gov/programs-surveys/popest/datasets/2010-2015/nation
al/asrh/nc-est2015-agesex-res.csv'
full_census_table = Table.read_table(census_url)

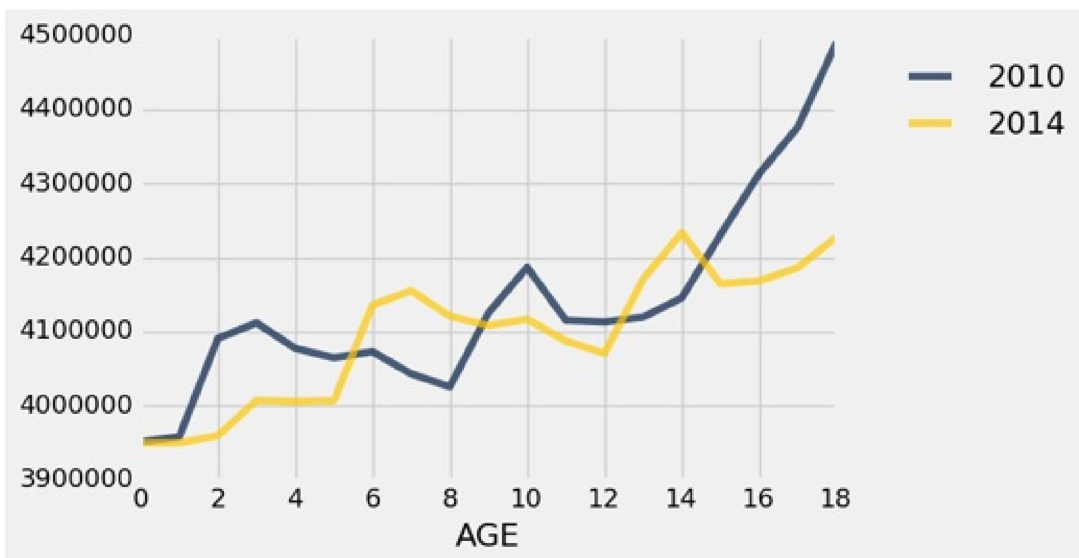
# Select columns from the full table and relabel some of them
partial_census_table = full_census_table.select(['SEX', 'AGE', 'POPESTIMATE2010', 'POP
ESTIMATE2014'])
us_pop = partial_census_table.relabeled('POPESTIMATE2010', '2010').relabeled('POPESTIM
ATE2014', '2014')

# Access the rows corresponding to all children, ages 0-18
children = us_pop.where('SEX', are.equal_to(0)).where('AGE', are.below(19)).drop('SEX'
)
children.show()
```

AGE	2010	2014
0	3951330	3949775
1	3957888	3949776
2	4090862	3959664
3	4111920	4007079
4	4077551	4005716
5	4064653	4006900
6	4073013	4135930
7	4043046	4155326
8	4025604	4120903
9	4125415	4108349
10	4187062	4116942
11	4115511	4087402
12	4113279	4070682
13	4119666	4171030
14	4145614	4233839
15	4231002	4164796
16	4313252	4168559
17	4376367	4186513
18	4491005	4227920

现在我们可以绘制两个叠加的线形图，显示 2010 年和 2014 年的不同年龄的儿童人数。方法调用类似于前面例子中的 `scatter` 调用。

```
children.plot('AGE')
```



在这个刻度上，重要的是要记住我们只有 `0,1,2` 岁等等的数据。 两个图形的点相互“交织”。

这些图表在一些地方相互交叉：例如，2010 年的 4 岁人数比 2014 年多，2014 年的 14 岁人数比 2010 年多。

当然，2014 年的 14 岁儿童大部分都是 2010 年的 10 岁儿童。为了看到这一点，请查看 14 岁的金色图表和 10 岁的蓝色图表。事实上，你会注意到，整个金色图表（2014 年）看起来像蓝色图表（2010 年）向右滑了 4 年。 由于 2010 年至 2014 年间进入该国的儿童的净效应，这个下滑幅度还是有所上升， 幸运的是，在这些年代，没有太多的生命损失。

条形图

对于本节的最后一个例子，我们看看加利福尼亚州以及整个美国的成人和儿童的种族分布情况。

凯撒家庭基金会根据人口普查数据，编制了美国人口种族分布情况。基金会的网站提供了 2014 年整个美国人口以及当年 18 岁以下的美国儿童的数据汇总。

这里是一个表格，采用了美国和加利福尼亚州的数据。 这些列代表美国和加利福尼亚州的每个人，美国和加州的儿童。 表格的主体包含不同类别的比例。 每一列显示了，该列对应的人群的种族分布。 所以在每一列中，条目总计为 1。

```
usa_ca = Table.read_table('usa_ca_2014.csv')
usa_ca
```

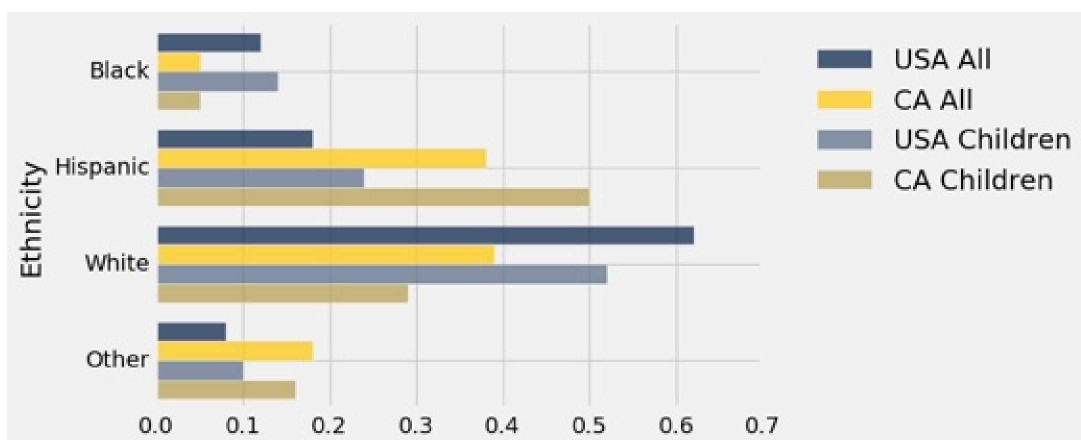
Ethnicity	USA All	CA All	USA Children	CA Children
Black	0.12	0.05	0.14	0.05
Hispanic	0.18	0.38	0.24	0.5
White	0.62	0.39	0.52	0.29
Other	0.08	0.18	0.1	0.16

我们自然想要比较这些分布。 直接比较列是有意义的，因为所有条目都是比例，因此在相同刻度上。

`barh` 方法允许我们通过在相同轴域上绘制多个条形图，将比较可视化。这个调用类似于 `scatter` 和 `plot` ：我们必须指定类别的公共轴。

> 译者注：轴域（Axes）是横轴和纵轴围城的区域。

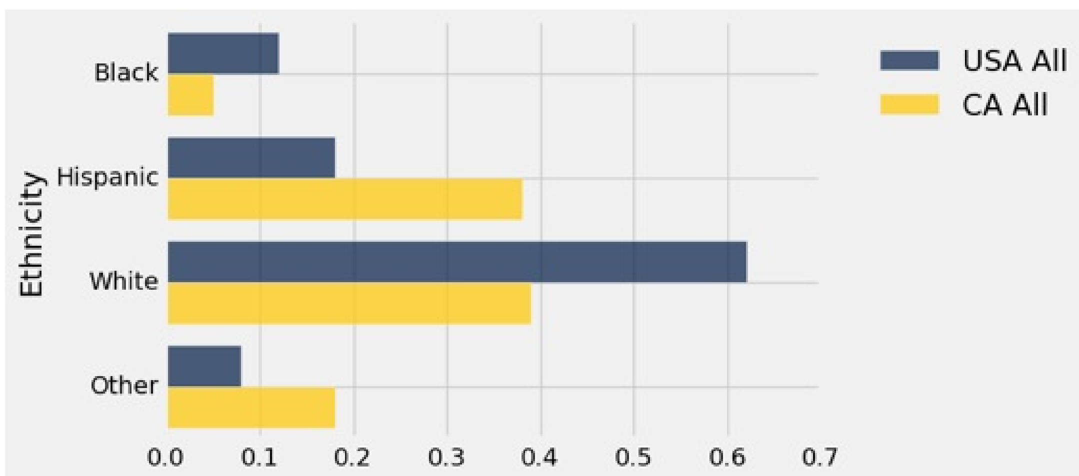
```
usa_ca.barh('Ethnicity')
```



虽然绘制叠加的条形图非常简单，但是我们可以在这个图表上找到太多的信息，以便能够理清种群之间的相似性和差异性。 似乎很清楚的是，美国所有人和美国儿童的种族分布比任何其他列都更相似，但是一次比较一对要容易得多。

首先比较美国和加利福尼亚的整个人口。

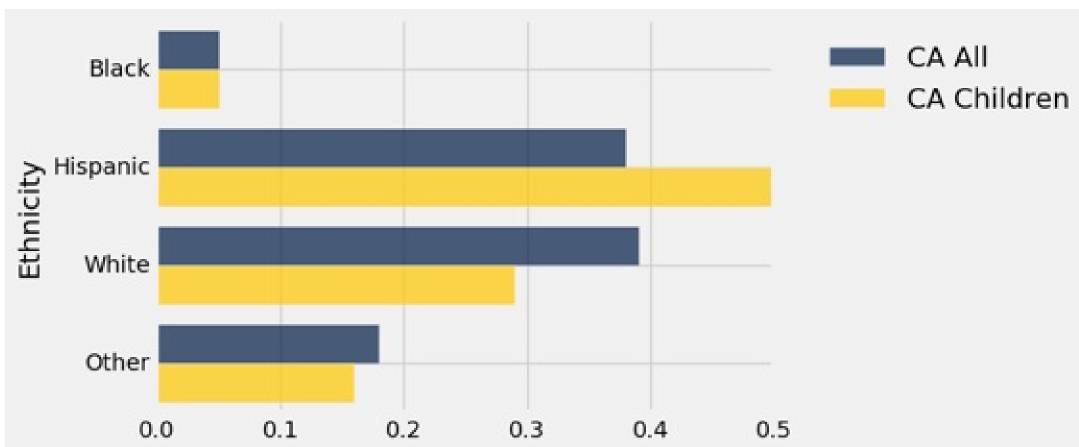
```
usa_ca.select('Ethnicity', 'USA All', 'CA All').barh('Ethnicity')
```



这两个分布是完全不同的。加利福尼亚州的拉美裔和其他类别比例较高，黑人和白人比例相应较低。这种差异主要是由于，加利福尼亚州的地理位置和移民模式，无论是历史上还是近几十年来。例如，加利福尼亚的“其他”类别包括相当一部分亚洲人和太平洋岛民。

从图中可以看出，2014 年加州近 40% 的人口是拉美裔。与该州儿童人口的比较表明，未来几年拉美裔人口的比例可能会更高。在加州儿童中，50% 属于拉美裔。

```
usa_ca.select('Ethnicity', 'CA All', 'CA Children').barh('Ethnicity')
```



更复杂的数据集自然会产生各种有趣的可视化效果，包括不同种类的重叠图形。为了分析这些数据，获得更多的数据操作技能的有帮助的，这样我们就可以将数据转化为一种形式，使我们能够使用本节中的方法。在下一章中，我们将介绍其中的一些技巧。

七、函数和表格

> 原文：Functions and Tables
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

通过使用 Python 中已有的函数，我们正在建立一个使用的技术清单，用于识别数据集中的规律和主题。 现在我们将探索Python编程语言的核心功能：函数定义。

我们在本书中已经广泛使用了函数，但从未定义过我们自己的函数。定义一个函数的目的是，给一个计算过程命名，它可能会使用多次。计算中有许多需要重复计算的情况。 例如，我们常常希望对表的列中的每个值执行相同的操作。

定义函数

`double` 函数的定义仅仅使一个数值加倍。

```
# Our first function definition

def double(x):
    """ Double x """
    return 2*x
```

我们通过编写 `def` 来开始定义任何函数。 下面是这个小函数的其他部分（语法）的细分：

Signature

- Calls to the function will look like this (with the same name and number of arguments). Example: double(3).
- When you call double, the argument can be any expression. (The name x doesn't affect calls.)
- In the body of the function, x is the name of the argument, as if the body included the code x = <the first argument>.

```
# Our first function definition

def double(x):
    """ Double x """
    return 2*x
```

Documentation ("docstring")

- Text that describes what the function does.
- Can be any string, traditionally triple-quoted so it can span several lines.
- Traditionally, the first line describes what the function does, briefly.
- Subsequent lines can give more detail and examples.
- Running double? will show this text, just like max? will show the documentation for the built-in function max.

Body

- All the code in here runs each time you call the function.
- The special statement return tells Python what the value of each call to this function is: it's the value of the expression after return.
- For example, the value of double(3) is 6. (Remember, when the argument is 3, it's like the body starts with x = 3.)
- Often, the body will have multiple lines of code that build up to computing the returned value. You can write any Python code here that you could write anywhere else.

Indentation

- Each line of code in the body is indented (that is, it's preceded by spaces).
- Traditionally, we use 2 or 4 spaces. They only need to be consistent.
- This tells Python that those lines are part of the body.
- The function's body ends at any unindented line.

当我们运行上面的单元格时，没有使特定的数字加倍，并且 `double` 主体中的代码还没有求值。因此，我们的函数类似于一个菜谱。 每次我们遵循菜谱中的指导，我们都需要以食材开始。 每次我们想用我们的函数来使一个数字加倍时，我们需要指定一个数字。

我们可以用和调用其他函数完全相同的方式，来调用 `double` 。 每次我们这样做的时候，主体中的代码都会执行，参数的值赋给了名称 `x` 。

```
double(17)
34
double(-0.6/4)
-0.3
```

以上两个表达式都是调用表达式。 在第二个里面，计算了表达式 `-0.6 / 4` 的值，然后将其作为参数 `x` 传递给 `double` 函数。 每个调用表达式最终都会执行 `double` 的主体，但使用不同的 `x` 值。

`double` 的主体只有一行：

```
return 2*x
```

执行这个 `return` 语句会完成 `double` 函数体的执行，并计算调用表达式的值。

`double` 的参数可以是任何表达式，只要它的值是一个数字。 例如，它可以是一个名称。 `double` 函数不知道或不在意如何计算或存储参数。 它唯一的工作是，使用传递给它的参数的值来执行它自己的主体。

```
any_name = 42
double(any_name)
84
```

参数也可以是任何可以加倍的值。例如，可以将整个数值数组作为参数传递给 `double` ，结果将是另一个数组。

```
double(make_array(3, 4, 5))
array([ 6,  8, 10])
```

但是，函数内部定义的名称（包括像double的x这样的参数）只存在一小会儿。 它们只在函数被调用的时候被定义，并且只能在函数体内被访问。 我们不能在 `double` 之外引用 `x` 。 技术术语是 `x` 具有局部作用域。

因此，即使我们在上面的单元格中调用了 `double` ，名称 `x` 也不能在函数体外识别。

```
x
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-401b30e3b8b5> in <module>()
----> 1 x

NameError: name 'x' is not defined
```

文档字符串。 虽然 `double` 比较容易理解，但是很多函数执行复杂的任务，并且没有解释就很难使用。 （你自己也可能已经发现了！）因此，一个组成良好的函数有一个唤起它的行为的名字，以及文档。 在 Python 中，这被称为文档字符串 - 描述了它的行为和对其参数的预期。 文档字符串也可以展示函数的示例调用，其中调用前面是 `>>>` 。

文档字符串可以是任何字符串，只要它是函数体中的第一个东西。 文档字符串通常在开始和结束处使用三个引号来定义，这允许字符串跨越多行。 第一行通常是函数的完整但简短的描述，而下面的行则为将来的用户提供了进一步的指导。

下面是一个名为 `percent` 的函数定义，它带有两个参数。定义包括一个文档字符串。

```
# A function with more than one argument

def percent(x, total):
    """Convert x to a percentage of total.

    More precisely, this function divides x by total,
    multiplies the result by 100, and rounds the result
    to two decimal places.

    >>> percent(4, 16)
    25.0
    >>> percent(1, 6)
    16.67
    """
    return round((x/total)*100, 2)
percent(33, 200)
16.5
```

将上面定义的函数 `percent` 与下面定义的函数 `percents` 进行对比。 后者以数组为参数，将数组中的所有数字转换为数组中所有值的百分数。 百分数都四舍五入到两位，这次使用 `round` 来代替 `np.round` ，因为参数是一个数组而不是一个数字。

```
def percents(counts):
    """Convert the values in array_x to percents out of the total of array_x."""
    total = counts.sum()
    return np.round((counts/total)*100, 2)
```

函数 `percents` 返回一个百分数的数组，除了四舍五入之外，它总计是 100。

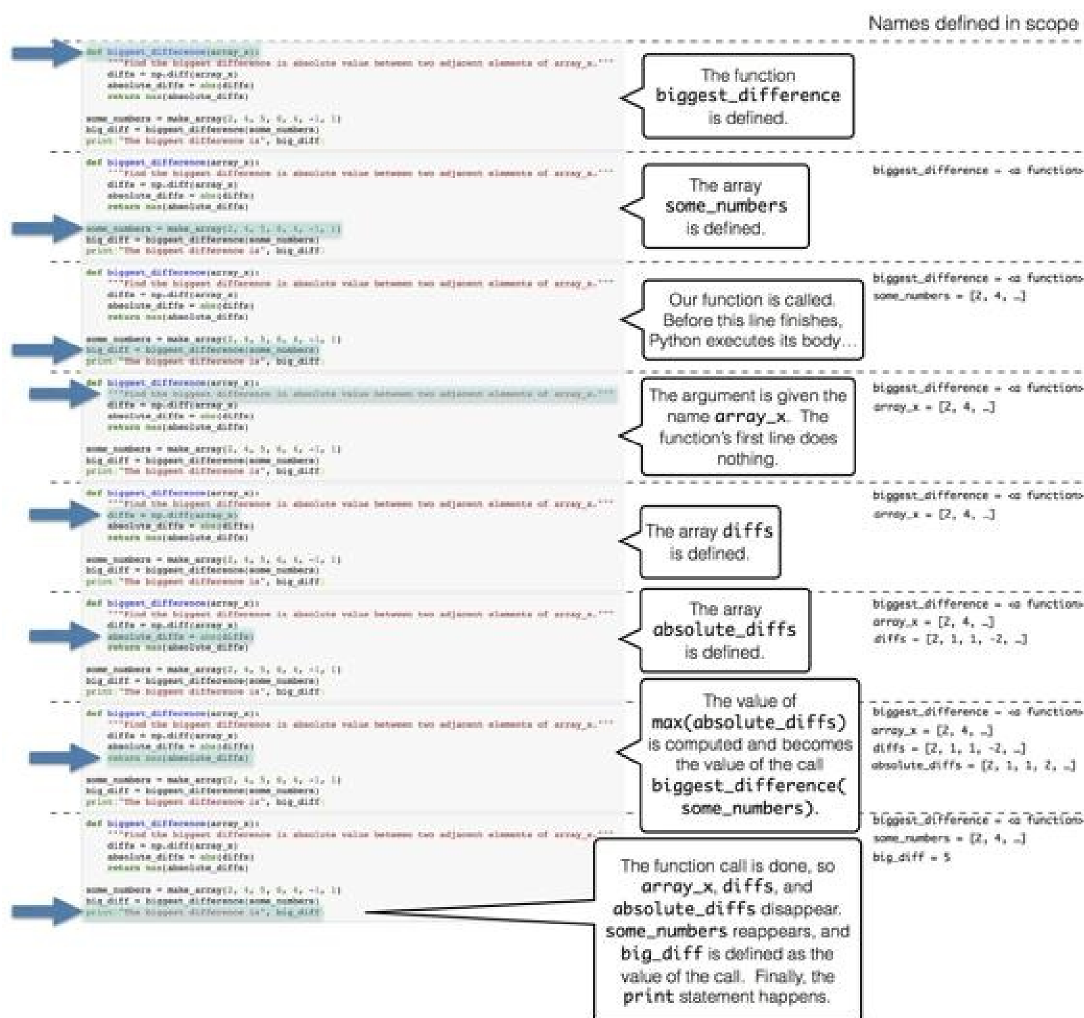
```
some_array = make_array(7, 10, 4)
percents(some_array)
array([ 33.33,  47.62,  19.05])
```

理解 Python 执行函数的步骤是有帮助的。 为了方便起见，我们在下面的同一个单元格中放入了函数定义和对这个函数的调用。

```
def biggest_difference(array_x):
    """Find the biggest difference in absolute value between two adjacent elements of array_x."""
    diffs = np.diff(array_x)
    absolute_diffs = abs(diffs)
    return max(absolute_diffs)

some_numbers = make_array(2, 4, 5, 6, 4, -1, 1)
big_diff = biggest_difference(some_numbers)
print("The biggest difference is", big_diff)
The biggest difference is 5
```

这就是当我们运行单元格时，所发生的事情。



多个参数

可以有多种方式来推广一个表达式或代码块，因此一个函数可以有多个参数，每个参数决定结果的不同方面。 例如，我们以前定义的百分比 `percents` ，每次都四舍五入到两位。 以下两个参的数定义允许不同调用四舍五入到不同的位数。

```
def percents(counts, decimal_places):
    """Convert the values in array_x to percents out of the total of array_x."""
    total = counts.sum()
    return np.round((counts/total)*100, decimal_places)

parts = make_array(2, 1, 4)
print("Rounded to 1 decimal place: ", percents(parts, 1))
print("Rounded to 2 decimal places:", percents(parts, 2))
print("Rounded to 3 decimal places:", percents(parts, 3))
Rounded to 1 decimal place:  [ 28.6   14.3   57.1]
Rounded to 2 decimal places: [ 28.57   14.29   57.14]
Rounded to 3 decimal places: [ 28.571   14.286   57.143]
```

这个新定义的灵活性来源于一个小的代价：每次调用该函数时，都必须指定小数位数。默认参数值允许使用可变数量的参数调用函数;在调用表达式中未指定的任何参数都被赋予其默认值，这在 `def` 语句的第一行中进行了说明。 例如，在 `percents` 的最终定义中，可选参

数 `decimal_places` 赋为默认值 `2` 。

```
def percents(counts, decimal_places=2):
    """Convert the values in array_x to percents out of the total of array_x."""
    total = counts.sum()
    return np.round((counts/total)*100, decimal_places)

parts = make_array(2, 1, 4)
print("Rounded to 1 decimal place:", percents(parts, 1))
print("Rounded to the default number of decimal places:", percents(parts))
Rounded to 1 decimal place: [ 28.6  14.3  57.1]
Rounded to the default number of decimal places: [ 28.57  14.29  57.14]
```

注：方法

函数通过将参数表达式放入函数名称后面的括号来调用。 任何独立定义的函数都是这样调用的。 你也看到了方法的例子，这些方法就像函数一样，但是用点符号来调用，比如 `some_table.sort(some_label)` 。 你定义的函数将始终首先使用函数名称，并传入所有参数来调用。

在列上应用函数

我们已经看到很多例子，通过将函数应用于现有列或其他数组，来创建新的表格的列。 所有这些函数都以数组作为参数。 但是我们经常打算，通过一个函数转换列中的条目，它不将数组作为它的函数。 例如，它可能只需要一个数字作为它的参数，就像下面定义的函数 `cut_off_at_100` 。

```
def cut_off_at_100(x):
    """The smaller of x and 100"""
    return min(x, 100)
cut_off_at_100(17)
17
cut_off_at_100(117)
100
cut_off_at_100(100)
100
```

如果参数小于或等于 100，函数 `cut_off_at_100` 只返回它的参数。但是如果参数大于 100，则返回 100。

在我们之前使用人口普查数据的例子中，我们看到变量 `AGE` 的值为 100，表示“100 岁以上”。以这种方式将年龄限制在 100 岁，正是 `cut_off_at_100` 所做的。

为了一次性对很多年龄使用这个函数，我们必须能够引用函数本身，而不用实际调用它。 类似地，我们可能会向厨师展示一个蛋糕的菜谱，并要求她用它来烤 6 个蛋糕。 在这种情况下，我们不会使用这个配方自己烘烤蛋糕， 我们的角色只是把菜谱给厨师。 同样，我们可以要求一个表格，在列中的 6 个不同的数字上调用 `cut_off_at_100` 。

首先，我们创建了一个表，一列是人，一列是它们的年龄。 例如，`C` 是 52 岁。

```
ages = Table().with_columns(
    'Person', make_array('A', 'B', 'C', 'D', 'E', 'F'),
    'Age', make_array(17, 117, 52, 100, 6, 101)
)
ages
```

Person	Age
A	17
B	117
C	52
D	100
E	6
F	101

应用

要在 100 岁截断年龄，我们将使用一个新的 `Table` 方法。 `apply` 方法在列的每个元素上调用一个函数，形成一个返回值的新数组。 为了指出要调用的函数，只需将其命名（不带引号或括号）。 输入值的列的名称必须是字符串，仍然出现在引号内。

```
ages.apply(cut_off_at_100, 'Age')
array([ 17, 100,  52, 100,   6, 100])
```

我们在这里所做的是，将 `cut_off_at_100` 函数应用于 `age` 表的 `Age` 列中的每个值。 输出是函数的相应返回值的数组。 例如，17 还是 17，117 变成了 100，52 还是 52，等等。

此数组的长度与 `age` 表中原始 `Age` 列的长度相同，可用作名为 `Cut Off Age` 的新列中的值，并与现有的 `Person` 和 `Age` 列共存。

```
ages.with_column(
    'Cut Off Age', ages.apply(cut_off_at_100, 'Age')
)
```

Person	Age	Cut Off Age
A	17	17
B	117	100
C	52	52
D	100	100
E	6	6
F	101	100

作为值的函数

我们已经看到,Python 有很多种值。 例如，`6` 是一个数值，`"cake"` 是一个文本值，`Table()` 是一个空表，`age` 是一个表值（因为我们在上面定义）的名称。

在 Python 中，每个函数（包括 `cut_off_at_100` ）也是一个值。 这有助于再次考虑菜谱。 蛋糕的菜谱是一个真实的东西，不同于蛋糕或配料，你可以给它一个名字，像“阿尼的蛋糕菜谱”。 当我们用 `def` 语句定义 `cut_off_at_100` 时，我们实际上做了两件事情：我们创建了一个函数来截断数字 100，我们给它命名为 `cut_off_at_100` 。

我们可以引用任何函数，通过写下它的名字，而没有实际调用它必需的括号或参数。当我们在上面调用 `apply` 时，我们做了这个。 当我们自己写下一个函数的名字，作为单元格中的最后一行时，Python 会生成一个函数的文本表示，就像打印一个数字或一个字符串值一样。

```
cut_off_at_100
<function __main__.cut_off_at_100>
```

请注意，我们没有使用引号（它只是一段文本）或 `cut_off_at_100()` （它是一个函数调用，而且是无效的）。我们只是写下 `cut_off_at_100` 来引用这个函数。

就像我们可以为其他值定义新名称一样，我们可以为函数定义新名称。 例如，假设我们想把我们的函数称为 `cut_off` ，而不是 `cut_off_at_100` 。 我们可以这样写：

```
cut_off = cut_off_at_100
```

现在 `cut_off` 就是函数名称了。它是 `cut_off_at_100` 的相同函数。所以打印出的值应该相同。

```
cut_off
<function __main__.cut_off_at_100>
```

让我们看看另一个 `apply` 的应用。

示例：预测

数据科学经常用来预测未来。 如果我们试图预测特定个体的结果 - 例如，她将如何回应处理方式，或者他是否会购买产品，那么将预测基于其他类似个体的结果是很自然的。

查尔斯·达尔文（Charles Darwin）的堂兄弗朗西斯·高尔顿（Sir Francis Galton）是使用这个思想来基于数值数据进行预测的先驱。 他研究了物理特征是如何传递下来的。

下面的数据是父母和他们的成年子女的身高测量值，由高尔顿仔细收集。 每行对应一个成年子女。 变量是家庭的数字代码，父母的身高（以英寸为单位），“双亲身高”，这是父母双方身高的加权平均值 [1]，家庭中子女的数量 ，以及子女的出生次序（第几个），性别和身高。

> [1] 高尔顿在计算男性和女性的平均身高之前，将女性身高乘上 1.08。对于这个的讨论，请查看 Chance，这是一个由美国统计协会出版的杂志。

```
# Galton's data on heights of parents and their adult children
galton = Table.read_table('galton.csv')
galton
```

family	father	mother	midparentHeight	children	childNum	gender
1	78.5	67	75.43	4	1	male
1	78.5	67	75.43	4	2	female
1	78.5	67	75.43	4	3	female
1	78.5	67	75.43	4	4	female
2	75.5	66.5	73.66	4	1	male
2	75.5	66.5	73.66	4	2	male
2	75.5	66.5	73.66	4	3	female
2	75.5	66.5	73.66	4	4	female
3	75	64	72.06	2	1	male
3	75	64	72.06	2	2	female

（省略了 924 行）

收集数据的主要原因是，能够预测父母所生的子女的成年身高，其中父母和数据集中的类似。让我们尝试这样做，用双亲的身高作为我们预测的基础变量。 因此双亲的身高是我们的预测性变量。

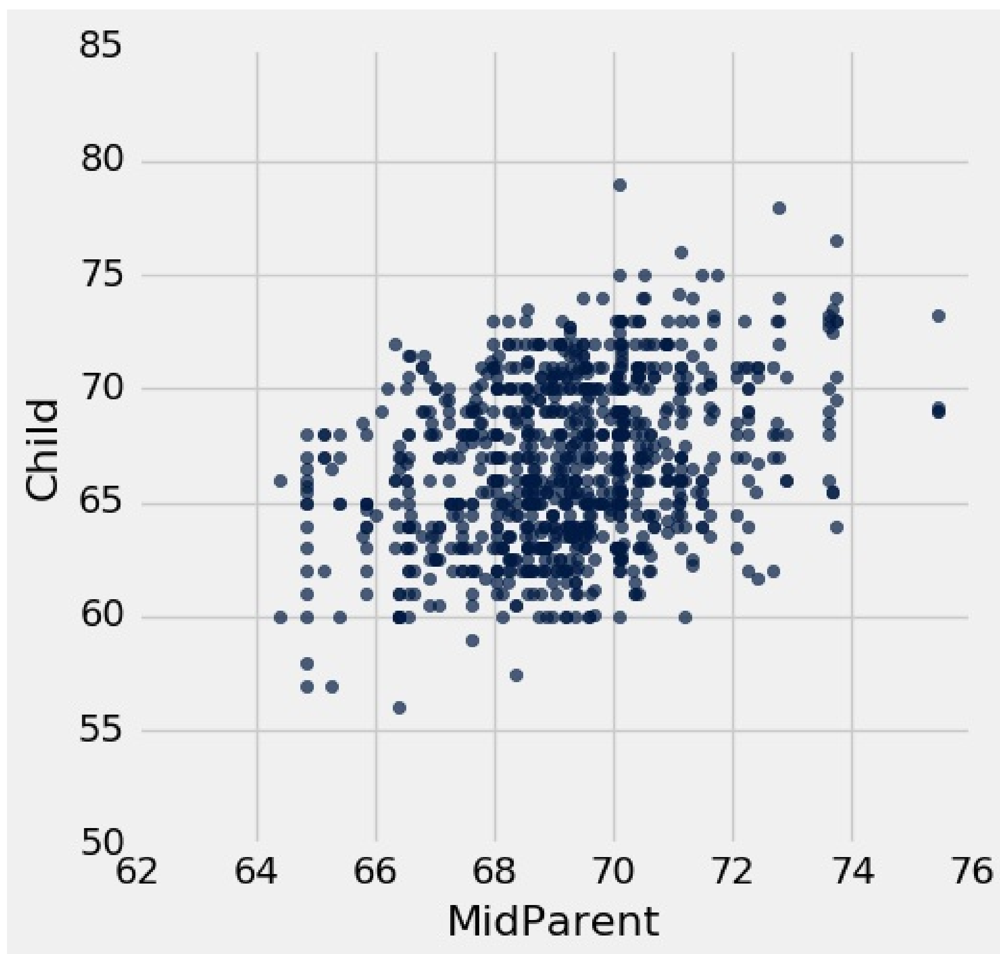
表格 `heights` 包含双亲和子女的身高。 两个变量的散点图显示了正相关，正如我们对这些变量的预期。

```
heights = galton.select(3, 7).relabeled(0, 'MidParent').relabeled(1, 'Child')
heights
```

MidParent	Child
75.43	73.2
75.43	69.2
75.43	69
75.43	69
73.66	73.5
73.66	72.5
73.66	65.5
73.66	65.5
72.06	71
72.06	68

（省略了 924 行）

```
heights.scatter(0)
```



现在假设高尔顿遇到了新的一对夫妇，与他的数据集类似，并且想知道他们的子女有多高。考虑到双亲身高是 68 英寸，他预测子女身高的一个好方法是什么？

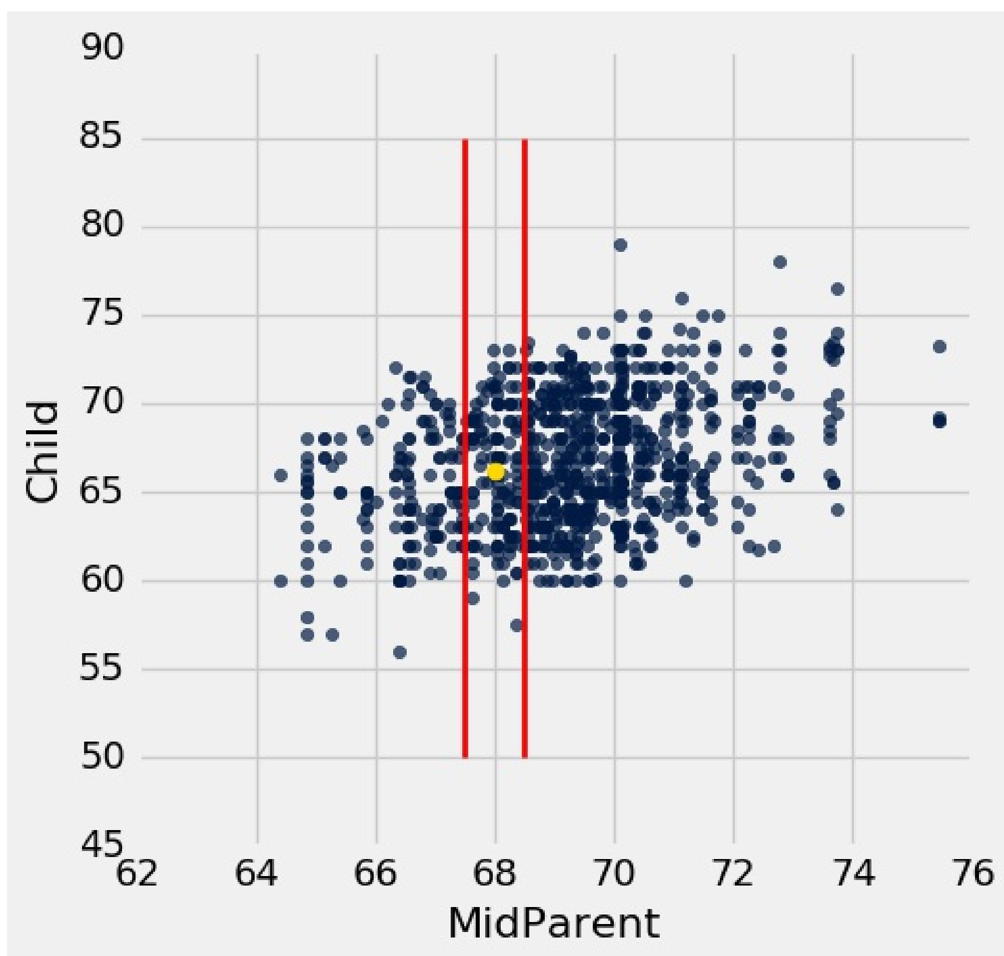
一个合理的方法是基于约 68 英寸的双亲身高对应的所有点，来做预测。预测值等于从这些点计算的子女身高的均值。

假设我们是高尔顿，并执行这个计划。现在我们只是对“68 英寸左右”的含义做一个合理的定义，并用它来处理。在课程的后面，我们将研究这种选择的后果。

我们的“接近”的意思是“在半英寸之内”。下图显示了 67.5 英寸和 68.5 英寸之间的双亲身高对应的所有点。这些都是红色直线之间的点。每一个点都对应一个子女；我们对新夫妇的子女身高的预测是所有子女的平均身高。这由金色的点表示。

忽略代码，仅仅专注于理解到达金色的点的心理过程。

```
heights.scatter('MidParent')
_ = plots.plot([67.5, 67.5], [50, 85], color='red', lw=2)
_ = plots.plot([68.5, 68.5], [50, 85], color='red', lw=2)
_ = plots.scatter(68, 66.24, color='gold', s=40)
```



为了准确计算出金色的点的位置，我们首先需要确定直线之间的所有点。 这些点对应于 `MidParent` 在 67.5 英寸和 68.5 英寸之间的行。

```
close_to_68 = heights.where('MidParent', are.between(67.5, 68.5))
close_to_68
```

MidParent	Child
68.44	62
67.94	71.2
67.94	67
68.33	62.5
68.23	73
68.23	72
68.23	69
67.98	73
67.98	71
67.98	71

（省略了 121 行）

双亲身高为 68 英寸的子女的预测身高，是这些行中子女的平均身高。 这是 66.24 英寸。

```
close_to_68.column('Child').mean()
66.24045801526718
```

我们现在有了一种方法，给定任何数据集中的双亲身高，就可以预测子女的身高。我们可以定义一个函数 `predict_child` 来实现它。 除了名称的选择之外，函数的主体由上面两个单元格中的代码组成。

```
def predict_child(mpht):
    """Predict the height of a child whose parents have a midparent height of mpht.

    The prediction is the average height of the children whose midparent height is
    in the range mpht plus or minus 0.5.
    """

    close_points = heights.where('MidParent', are.between(mpht-0.5, mpht + 0.5))
    return close_points.column('Child').mean()
```

给定 68 英寸的双亲身高，函数 `predict_child` 返回与之前相同的预测（66.24 英寸）。 定义函数的好处在于，我们可以很容易地改变预测变量的值，并得到一个新的预测结果。

```
predict_child(68)
66.24045801526718
predict_child(74)
70.415789473684214
```

这些预测有多好？ 我们可以了解它，通过将预测值与我们已有的数据进行比较。 为此，我们首先将函数 `predict_child` 应用于 `Midparent` 列，并将结果收入称为 `Prediction` 的新列中。

```
# Apply predict_child to all the midparent heights

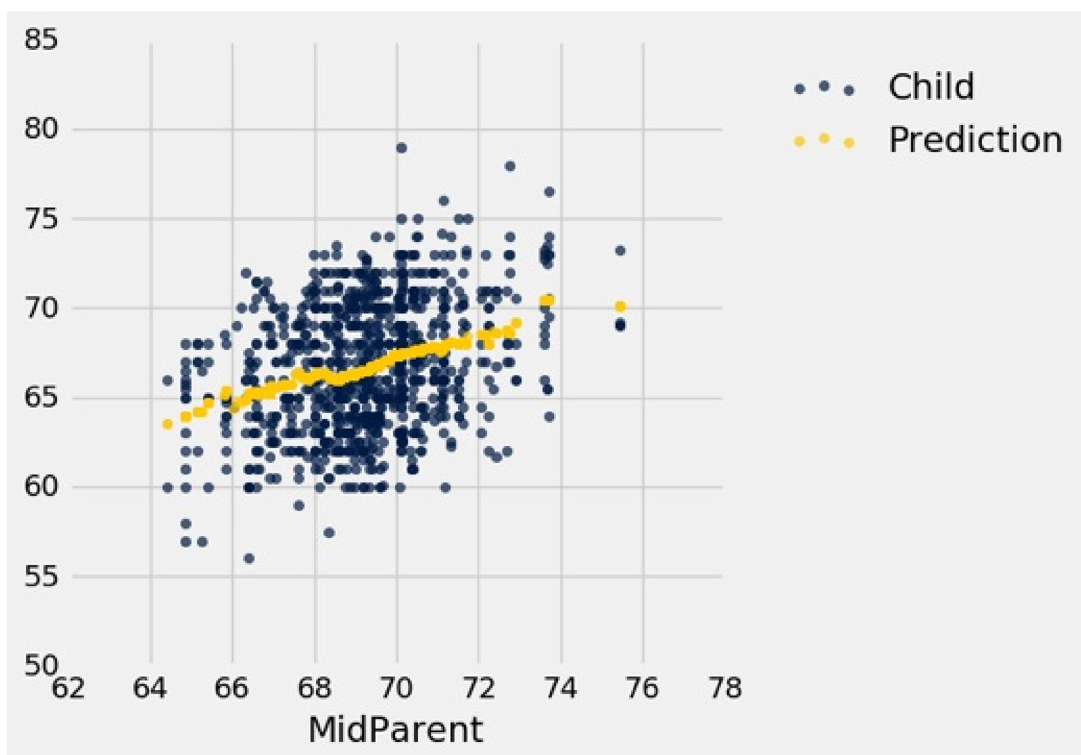
heights_with_predictions = heights.with_column(
    'Prediction', heights.apply(predict_child, 'MidParent')
)
heights_with_predictions
```

MidParent	Child	Prediction
75.43	73.2	70.1
75.43	69.2	70.1
75.43	69	70.1
75.43	69	70.1
73.66	73.5	70.4158
73.66	72.5	70.4158
73.66	65.5	70.4158
73.66	65.5	70.4158
72.06	71	68.5025
72.06	68	68.5025

（省略了 924 行）

为了查看预测值相对于观察数据的位置，可以使用 `MidParent` 作为公共水平轴绘制重叠的散点图。

```
heights_with_predictions.scatter('MidParent')
```



金色的点的图形称为均值图，因为每个金色的点都是两条直线的中心，就像之前绘制的那样。每个都按照给定的双亲高度，做出了子女高度的预测。例如，散点图显示，对于 72 英寸的双亲高度，子女的预测高度将在 68 英寸和 69 英寸之间，事实上，`predict_child(72)` 返回

68.5。

高尔顿的计算和可视化与我们非常相似，除了他没有 Python。他通过散点图绘制了均值图，并注意到它大致沿着直线。这条直线现在被称为回归线，是最常见的预测方法之一。高尔顿的朋友，数学家卡尔·皮尔森（Karl Pearson）用这些分析来形式化关联的概念。

这个例子，就像约翰·斯诺（John Snow）对霍乱死亡的分析一样，说明了现代数据科学的一些基本概念的根源可追溯到一个多世纪之前。高尔顿的方法，比如我们在这里使用的方法，是最近邻预测方法的雏形，现在在不同的环境中有着有效的应用。机器学习的现代领域包括这些方法的自动化，来基于庞大且快速发展的数据集进行预测。

按照单变量分类

数据科学家经常需要根据共有的特征，将个体分成不同的组，然后确定组的一些特征。 例如，在使用高尔顿高度数据的例子中，我们看到根据父母的平均高度对家庭进行分类，然后找出每个小组中子女的平均身高，较为实用。

这部分关于将个体分类到非数值类别。我们从回顾 `gourp` 的基本用法开始。

计算每个分类的数量

具有单个参数的 `group` 方法计算列中每个值的数量。 结果中，分组列（用于分组的列）中的每个唯一值是一行。

这是一个关于冰淇淋圆通的小型数据表。 `group` 方法可以用来列出不同的口味，并提供每种口味的计数。

```
cones = Table().with_columns(
    'Flavor', make_array('strawberry', 'chocolate', 'chocolate', 'strawberry', 'chocol
ate'),
    'Price', make_array(3.55, 4.75, 6.55, 5.25, 5.25)
)
cones
```

Flavor	Price
strawberry	3.55
chocolate	4.75
chocolate	6.55
strawberry	5.25
chocolate	5.25

```
cones.group('Flavor')
```

Flavor	count
chocolate	3
strawberry	2

有两个不同的类别，巧克力和草莓。 `group` 的调用会在每个类别中创建一个计数表。 该列默认称为 `count` ，并包含每个类别中的行数。

注意，这一切都可以从 `Flavor` 列中找到。 `Price` 列尚未使用。

但是如果我们想要每种不同风味的圆筒的总价格呢？ 这是 `group` 的第二个参数的作用。

发现每个类别的特征

`group` 的可选的第二个参数是一个函数，用于聚合所有这些行的其他列中的值。 例如， `sum` 将累计与每个类别匹配的所有行中的价格。 这个结果中，分组列中每个唯一值是一行，但与原始表列数相同。

为了找到每种口味的总价格，我们再次调用 `group` ，用 `Flavor` 作为第一个参数。 但这一次有第二个参数：函数名称 `sum` 。

```
cones.group('Flavor', sum)
```

Flavor	Price sum
chocolate	16.55
strawberry	8.8

为了创建这个新表格， `group` 已经计算了对应于每种不同口味的，所有行中的 `Price` 条目的总和。 三个 `chocolate` 行的价格共计 `$16.55` （你可以假设价格是以美元计量的）。 两个 `strawberry` 行的价格共计 `8.80` 。

新创建的“总和”列的标签是 `Price sum` ，它通过使用被求和列的标签，并且附加单词 `sum` 创建。

由于 `group` 计算除了类别之外的所有列的 `sum` ，因此不需要指定必须对价格求和。

为了更详细地了解 `group` 在做什么，请注意，你可以自己计算总价格，不仅可以通过心算，还可以使用代码。 例如，要查找所有巧克力圆筒的总价格，你可以开始创建一个仅包含巧克力圆筒的新表，然后访问价格列：

```
cones.where('Flavor', are.equal_to('chocolate')).column('Price')
array([ 4.75,  6.55,  5.25])
sum(cones.where('Flavor', are.equal_to('chocolate')).column('Price'))
16.550000000000001
```

这就是 `group` 对 `Flavor` 中每个不同的值所做的事情。

```
# For each distinct value in `Flavor, access all the rows
# and create an array of `Price`

cones_choc = cones.where('Flavor', are.equal_to('chocolate')).column('Price')
cones_strawb = cones.where('Flavor', are.equal_to('strawberry')).column('Price')

# Display the arrays in a table

grouped_cones = Table().with_columns(
    'Flavor', make_array('chocolate', 'strawberry'),
    'Array of All the Prices', make_array(cones_choc, cones_strawb)
)

# Append a column with the sum of the `Price` values in each array

price_totals = grouped_cones.with_column(
    'Sum of the Array', make_array(sum(cones_choc), sum(cones_strawb))
)
price_totals
```

Flavor	Array of All the Prices	Sum of the Array
chocolate	[4.75 6.55 5.25]	16.55
strawberry	[3.55 5.25]	8.8

你可以用任何其他可以用于数组的函数来替换 `sum` 。例如，你可以使用 `max` 来查找每个类别中的最大价格：

```
cones.group('Flavor', max)
```

Flavor	Price max
chocolate	6.55
strawberry	5.25

同样， `group` 在每个 `Flavor` 分类中创建价格数组，但现在它寻找每个数组的 `max` 。

```
price_maxes = grouped_cones.with_column(
    'Max of the Array', make_array(max(cones_choc), max(cones_strawb))
)
price_maxes
```

Flavor	Array of All the Prices	Max of the Array
chocolate	[4.75 6.55 5.25]	6.55
strawberry	[3.55 5.25]	5.25

实际上，只有一个参数的原始调用，与使用 `len` 作为函数并清理表格的效果相同。

```
lengths = grouped_cones.with_column(
    'Length of the Array', make_array(len(cones_choc), len(cones_strawb))
)
lengths
```

Flavor	Array of All the Prices	Length of the Array
chocolate	[4.75 6.55 5.25]	3
strawberry	[3.55 5.25]	2

示例：NBA 薪水

`nba` 表包含了 2015~2016 年 NBA 球员的数据。 我们早些时候审查了这些数据。 回想一下，薪水以百万美元计算。

```
nba1 = Table.read_table('nba_salaries.csv')
nba = nba1.relabeled("'15-'16 SALARY", 'SALARY')
nba
```

PLAYER	POSITION	TEAM	SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4
Mike Scott	PF	Atlanta Hawks	3.33333
Kent Bazemore	SF	Atlanta Hawks	2
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452

（省略了 407 行）

（1）每支球队为球员的工资支付了多少钱？

唯一涉及的列是 `TEAM` 和 `SALARY` 。 我们必须按 `TEAM` 对这些行进行分组，然后对这些分类的工资进行求和。

```
teams_and_money = nba.select('TEAM', 'SALARY')
teams_and_money.group('TEAM', sum)
```

TEAM	SALARY sum
Atlanta Hawks	69.5731
Boston Celtics	50.2855
Brooklyn Nets	57.307
Charlotte Hornets	84.1024
Chicago Bulls	78.8209
Cleveland Cavaliers	102.312
Dallas Mavericks	65.7626
Denver Nuggets	62.4294
Detroit Pistons	42.2118
Golden State Warriors	94.0851

（省略了 20 行）

（2）五个位置的每个中有多少个 NBA 球员呢？

我们必须按 `POSITION` 分类并计数。 这可以通过一个参数来完成：

```
nba.group('POSITION')
```

POSITION	count
C	69
PF	85
PG	85
SF	82
SG	96

（3）五个位置的每个中，球员平均薪水是多少？

这一次，我们必须按 `POSITION` 分组，并计算薪水的均值。 为了清楚起见，我们将用一张表格来描述位置和薪水。

```
positions_and_money = nba.select('POSITION', 'SALARY')
positions_and_money.group('POSITION', np.mean)
```

POSITION	SALARY mean
C	6.08291
PF	4.95134
PG	5.16549
SF	5.53267
SG	3.9882

中锋是最高薪的职位，均值超过 600 万美元。

如果我们开始没有选择这两列，那么 `group` 不会尝试对 `nba` 中的类别列计算“平均”。（“亚特兰大老鹰”和“波士顿凯尔特人队”这两个字符串是不可能平均）。它只对数值列做算术，其余的都是空白的。

```
nba.group('POSITION', np.mean)
```

POSITION	PLAYER mean	TEAM mean	SALARY mean
C			6.08291
PF			4.95134
PG			5.16549
SF			5.53267
SG			3.9882

交叉分类

通过多个变量的交叉分类

当个体具有多个特征时，有很多不同的对他们分类的方法。 例如，如果我们有大学生的人口数据，对于每个人我们都有专业和大学的年数，那么这些学生就可以按照专业，按年份，或者是专业和年份的组合来分类。

`group` 方法也允许我们根据多个变量划分个体。 这被称为交叉分类。

两个变量：计算每个类别偶对的数量

`more_cones` 表记录了六个冰淇淋圆筒的味道，颜色和价格。

```
more_cones = Table().with_columns(
    'Flavor', make_array('strawberry', 'chocolate', 'chocolate', 'strawberry', 'chocol
ate', 'bubblegum'),
    'Color', make_array('pink', 'light brown', 'dark brown', 'pink', 'dark brown', 'pi
nk'),
    'Price', make_array(3.55, 4.75, 5.25, 5.25, 5.25, 4.75)
)

more_cones
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
chocolate	dark brown	5.25
bubblegum	pink	4.75

我们知道如何使用`group`，来计算每种口味的冰激凌圆筒的数量。

```
more_cones.group('Flavor')
```

Flavor	count
bubblegum	1
chocolate	3
strawberry	2

但是现在每个圆筒也有一个颜色。 为了将圆筒按风味和颜色进行分类，我们将把标签列表作为参数传递给`group`。 在分组列中出现的每个唯一值的组合，在生成的表格中都占一行。 和以前一样，一个参数（这里是一个列表，但是也可以是一个数组）提供了行数。

虽然有六个圆筒，但只有四种风味和颜色的唯一组合。 两个圆筒是深褐色的巧克力，还有两个粉红色的草莓。

```
more_cones.group(['Flavor', 'Color'])
```

Flavor	Color	count
bubblegum	pink	1
chocolate	dark brown	2
chocolate	light brown	1
strawberry	pink	2

两个变量：查找每个类别偶对的特征

第二个参数聚合所有其他列，它们不在分组列的列表中。

```
more_cones.group(['Flavor', 'Color'], sum)
```

Flavor	Color	Price sum
bubblegum	pink	4.75
chocolate	dark brown	10.5
chocolate	light brown	4.75
strawberry	pink	8.8

三个或更多的变量。 你可以使用 `group` ，按三个或更多类别变量对行分类。 只要将它们全部包含列表中，它是第一个参数。 但是由多个变量交叉分类可能会变得复杂，因为不同类别组合的数量可能相当大。

数据透视表：重新排列 `group` 的输出

交叉分类的许多使用只涉及两个类别变量，如上例中的 `Flavor` 和 `Color` 。 在这些情况下，可以在不同类型的表中显示分类结果，称为数据透视表（pivot table）。 数据透视表，也被称为列联表（contingency table），可以更容易地处理根据两个变量进行分类的数据。

回想一下，使用 `group` 来计算每个风味和颜色的类别偶对的圆筒数量：

```
more_cones.group(['Flavor', 'Color'])
```

Flavor	Color	count
bubblegum	pink	1
chocolate	dark brown	2
chocolate	light brown	1
strawberry	pink	2

使用 `Table` 的 `pivot` 方法可以以不同方式展示相同数据。暂时忽略这些代码，然后查看所得表。

```
more_cones.pivot('Flavor', 'Color')
```

Color	bubblegum	chocolate	strawberry
dark brown	0	2	0
light brown	0	1	0
pink	1	0	2

请注意，此表格显示了所有九种可能的风味和颜色偶对，包括我们的数据中不存在的偶对，比如“深棕色泡泡糖”。 还要注意，每个偶对中的计数都出现在表格的正文中：要找到浅棕色巧克力圆筒的数量，用眼睛沿着浅棕色的行看，直到它碰到巧克力一列。

`group` 方法接受两个标签的列表，因为它是灵活的：可能需要一个或三个或更多。 另一方面，数据透视图总是需要两个列标签，一个确定列，一个确定行。

`pivot` 方法与 `group` 方法密切相关： `group` 将拥有相同值的组合的行分组在一起。它与 `group` 不同，因为它将所得值组织在一个网格中。 `pivot` 的第一个参数是列标签，包含的值将用于在结果中形成新的列。第二个参数是用于行的列标签。结果提供了原始表的所有行的计数，它们拥有相同的行和列值组合。

像 `group` 一样， `pivot` 可以和其他参数一同使用，来发现每个类别组合的特征。名为 `values` 的第三个可选参数表示一列值，它们替换网格的每个单元格中的计数。所有这些值将不会显示，但是；第四个参数 `collect` 表示如何将它们全部汇总到一个聚合值中，来显示在单元格中。

用例子来澄清这一点。这里是一个透视表，用于寻找每个单元格中的圆筒的总价格。

```
more_cones.pivot('Flavor', 'Color', values='Price', collect=sum)
```

Color	bubblegum	chocolate	strawberry
dark brown	0	10.5	0
light brown	0	4.75	0
pink	4.75	0	8.8

这里 `group` 做了同一件事。

```
more_cones.group(['Flavor', 'Color'], sum)
```

Flavor	Color	Price sum
bubblegum	pink	4.75
chocolate	dark brown	10.5
chocolate	light brown	4.75
strawberry	pink	8.8

尽管两个表中的数字都相同，但由`pivot`生成的表格更易于阅读，因而更易于分析。 透视表的优点是它将分组的值放到相邻的列中，以便它们可以进行组合和比较。

示例：加州成人的教育和收入

加州的开放数据门户是丰富的加州生活的信息来源。 这是 2008 至 2014 年间加利福尼亚州教育程度和个人收入的数据集。数据来源于美国人口普查的当前人口调查。

对于每年，表格都记录了加州的`Population Count`（人口数量），按照年龄，性别，教育程度和个人收入，构成不同的组合。 我们将只研究 2014 年的数据。

```
full_table = Table.read_table('educ_inc.csv')
ca_2014 = full_table.where('Year', are.equal_to('1/1/14 0:00')).where('Age', are.not_e
qual_to('00 to 17'))
ca_2014
```

Year	Age	Gender	Educational Attainment	Personal Income	Population Count
1/1/14 0:00	18 to 64	Female	No high school diploma	H: 75,000 and over	2058
1/1/14 0:00	65 to 80+	Male	No high school diploma	H: 75,000 and over	2153
1/1/14 0:00	65 to 80+	Female	No high school diploma	G: 50,000 to 74,999	4666
1/1/14 0:00	65 to 80+	Female	High school or equivalent	H: 75,000 and over	7122
1/1/14 0:00	65 to 80+	Female	No high school diploma	F: 35,000 to 49,999	7261
1/1/14 0:00	65 to 80+	Male	No high school diploma	G: 50,000 to 74,999	8569
1/1/14 0:00	18 to 64	Female	No high school diploma	G: 50,000 to 74,999	14635
1/1/14 0:00	65 to 80+	Male	No high school diploma	F: 35,000 to 49,999	15212
1/1/14 0:00	65 to 80+	Male	College, less than 4-yr degree	B: 5,000 to 9,999	15423
1/1/14 0:00	65 to 80+	Female	Bachelor's degree or higher	A: 0 to 4,999	15459

（省略了 117 行）

表中的每一行对应一组年龄，性别，教育程度和收入。 总共有 127 个这样的组合！

作为第一步，从一个或两个变量开始是个好主意。 我们只关注一对：教育程度和个人收入。

```
educ_inc = ca_2014.select('Educational Attainment', 'Personal Income', 'Population Cou
nt')
educ_inc
```

Educational Attainment	Personal Income	Population Count
No high school diploma	H: 75,000 and over	2058
No high school diploma	H: 75,000 and over	2153
No high school diploma	G: 50,000 to 74,999	4666
High school or equivalent	H: 75,000 and over	7122
No high school diploma	F: 35,000 to 49,999	7261
No high school diploma	G: 50,000 to 74,999	8569
No high school diploma	G: 50,000 to 74,999	14635
No high school diploma	F: 35,000 to 49,999	15212
College, less than 4-yr degree	B: 5,000 to 9,999	15423
Bachelor's degree or higher	A: 0 to 4,999	15459

（省略了 117 行）

我们先看看教育程度。这个变量的分类已经由不同的收入水平细分了。因此，我们将按照教育程度分组，并将每个分类中的人口数量相加。

```
education = educ_inc.select('Educational Attainment', 'Population Count')
educ_totals = education.group('Educational Attainment', sum)
educ_totals
```

Educational Attainment	Population Count sum
Bachelor's degree or higher	8525698
College, less than 4-yr degree	7775497
High school or equivalent	6294141
No high school diploma	4258277

教育程度只有四类。计数太大了，查看百分比更有帮助。为此，我们将使用前面章节中定义的函数 `percents`。它将数值数组转换为输入数组总量的百分比数组。

```
def percents(array_x):
    return np.round( (array_x/sum(array_x))*100, 2)
```

我们现在有加州成人的教育程度分布。超过 30% 的人拥有学士或更高学位，而几乎 16% 没有高中文凭。

```
educ_distribution = educ_totals.with_column(
    'Population Percent', percents(educ_totals.column(1))
)
educ_distribution
```

Educational Attainment	Population Count sum	Population Percent
Bachelor's degree or higher	8525698	31.75
College, less than 4-yr degree	7775497	28.96
High school or equivalent	6294141	23.44
No high school diploma	4258277	15.86

通过使用 `pivot` ，我们可以得到一张加州成人的透视表（计数表），按照 `Educational Attainment` 和 `Personal Income` 交叉分类。

```
totals = educ_inc.pivot('Educational Attainment', 'Personal Income', values='Population Count', collect=sum)
totals
```

Personal Income	Bachelor's degree or higher	College, less than 4-yr degree	High school or equivalent	No high school diploma
A: 0 to 4,999	575491	985011	1161873	1204529
B: 5,000 to 9,999	326020	810641	626499	597039
C: 10,000 to 14,999	452449	798596	692661	664607
D: 15,000 to 24,999	773684	1345257	1252377	875498
E: 25,000 to 34,999	693884	1091642	929218	464564
F: 35,000 to 49,999	1122791	1112421	782804	260579
G: 50,000 to 74,999	1594681	883826	525517	132516
H: 75,000 and over	2986698	748103	323192	58945

在这里你可以看到 `pivot` 相比其他方法的威力。 计数的每一列都是个人收入在特定教育程度中的分布。 将计数转换为百分数可以让我们比较四个分布。

```
distributions = totals.select(0).with_columns(
    "Bachelor's degree or higher", percents(totals.column(1)),
    'College, less than 4-yr degree', percents(totals.column(2)),
    'High school or equivalent', percents(totals.column(3)),
    'No high school diploma', percents(totals.column(4))
    )

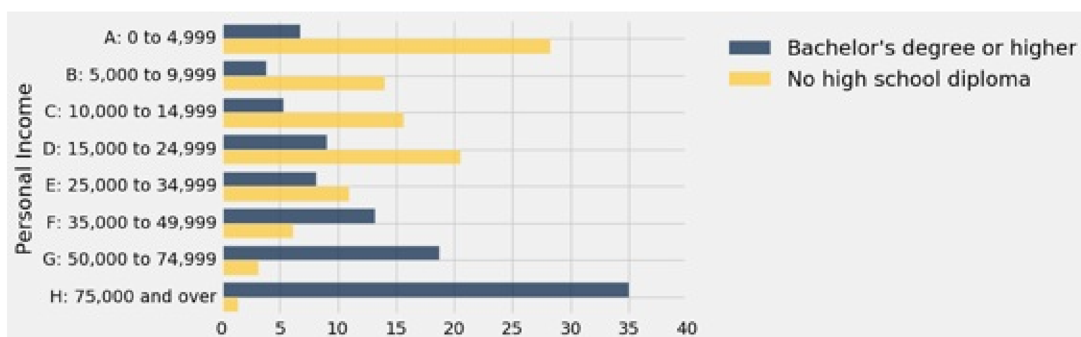
distributions
```

Personal Income	Bachelor's degree or higher	College, less than 4-yr degree	High school or equivalent	No high school diploma
A: 0 to 4,999	6.75	12.67	18.46	28.29
B: 5,000 to 9,999	3.82	10.43	9.95	14.02
C: 10,000 to 14,999	5.31	10.27	11	15.61
D: 15,000 to 24,999	9.07	17.3	19.9	20.56
E: 25,000 to 34,999	8.14	14.04	14.76	10.91
F: 35,000 to 49,999	13.17	14.31	12.44	6.12
G: 50,000 to 74,999	18.7	11.37	8.35	3.11
H: 75,000 and over	35.03	9.62	5.13	1.38

一眼就能看出，超过 35% 的学士或以上学位的收入达到 $75,000 美元以上，而其他教育分类中，少于 10% 的人达到了这一水平。

下面的条形图比较了没有高中文凭的加州成年人的个人收入分布情况，和完成学士或更高学位的人的收入分布情况。分布的差异是惊人的。教育程度与个人收入有明显的正相关关系。

```
distributions.select(0, 1, 4).barh(0)
```



按列连接表

通常，同一个人的数据在多个表格中维护。 例如，大学的一个办公室可能有每个学生完成学位的时间的数据，而另一个办公室则有学生学费和经济援助的数据。

为了了解学生的经历，将两个数据集放在一起可能会有帮助。 如果数据是在两个表中，每个学生都有一行，那么我们希望将这些列放在一起，确保行是匹配的，以便将每个学生的信息保持在一行上。

让我们在一个简单的示例的背景下实现它，然后在更大的数据集上使用这个方法。

圆筒表是我们以前遇到的。 现在假设每种口味的冰激凌的评分都在独立的表格中。

```
cones = Table().with_columns(
    'Flavor', make_array('strawberry', 'vanilla', 'chocolate', 'strawberry', 'chocolat
e'),
    'Price', make_array(3.55, 4.75, 6.55, 5.25, 5.75)
)
cones
```

Flavor	Price
strawberry	3.55
vanilla	4.75
chocolate	6.55
strawberry	5.25
chocolate	5.75

```
ratings = Table().with_columns(
    'Kind', make_array('strawberry', 'chocolate', 'vanilla'),
    'Stars', make_array(2.5, 3.5, 4)
)
ratings
```

Kind	Stars
strawberry	2.5
chocolate	3.5
vanilla	4

每个表都有一个包含冰淇淋风味的列：`cones` 有 `Flavor` 列，`ratings` 有 `Kind` 列。 这些列中的条目可以用来连接两个表。

`join` 方法创建一个新的表，其中 `cones` 表中的每个圆筒都增加了评分信息。 对于 `cones` 中的每个圆筒，`join` 会找到 `ratings` 中的行，它的 `Kind` 匹配圆筒的 `Flavor` 。 我们必须告诉 `join` 使用这些列进行匹配。

```
rated = cones.join('Flavor', ratings, 'Kind')
rated
```

Flavor	Price	Stars
chocolate	6.55	3.5
chocolate	5.75	3.5
strawberry	3.55	2.5
strawberry	5.25	2.5
vanilla	4.75	4

现在每个圆筒不仅拥有价格，而且还有风味评分。

一般来说，使用来自另一个表（比如 `table2` ）的信息来扩充一个表（如 `table1` ）的 `join` 调用如下所示：

```
table1.join(table1_column_for_joining, table2, table2_column_for_joining)
```

新的 `rated` 表使我们能够计算出价格与星星的比值，你可以把它看作是非正式的价值衡量标准。 低的值更好 - 它们意味着你为评分的每个星星花费更少。

```
rated.with_column('$/Star', rated.column('Price') / rated.column('Stars')).sort(3)
```

Flavor	Price	Stars	$/Star
vanilla	4.75	4	1.1875
strawberry	3.55	2.5	1.42
chocolate	5.75	3.5	1.64286
chocolate	6.55	3.5	1.87143
strawberry	5.25	2.5	2.1

虽然草莓在这三种口味中评分最低，但是这个标准下草莓更加便宜，因为每颗星星的花费并不高。

警告。顺序很重要。由于 `join` 中的第二个表用于扩充第一个表，所以重要的是，第一个表中的每一行在第二个表中只有一个匹配的行。如果第一个表中的某一行在第二个表中没有匹配项，则信息可能丢失。如果第一个表中的某一行在第二个表中有多个匹配项，那么 `join` 将只选择一个，这也是一种信息丢失。

我们可以在下面的例子中看到它，它试图通过相同的两列连接相同的两个表格，但是以另一种顺序。这种连接是没有意义的：它试图用价格来扩展每种风味的评分，但是根据 `cones` 表，每种风味都有一个以上的圆筒（和价格）。结果是两个圆筒消失了。`join` 方法仅仅在 `cones` 寻找对应 `chocolate` 的第一行，而忽略其他行。

```
ratings.join('Kind', cones, 'Flavor')
```

Kind	Stars	Price
chocolate	3.5	6.55
strawberry	2.5	3.55
vanilla	4	4.75

假设有个冰淇淋的评分表，我们已经求出了每种风味的平均评分。

```
reviews = Table().with_columns(
    'Flavor', make_array('vanilla', 'chocolate', 'vanilla', 'chocolate'),
    'Stars', make_array(5, 3, 5, 4)
)
reviews
```

Flavor	Stars
vanilla	5
chocolate	3
vanilla	5
chocolate	4

```
average_review = reviews.group('Flavor', np.average)
average_review
```

Flavor	Stars average
chocolate	3.5
vanilla	5

我们可以连接 `cones` 和 `average_review`，通过提供用于连接的列标签。

```
cones.join('Flavor', average_review, 'Flavor')
```

Flavor	Price	Stars average
chocolate	6.55	3.5
chocolate	5.75	3.5
vanilla	4.75	5

注意草莓圆筒是如何消失的。 没有草莓圆筒的评价，所以没有草莓的行可以连接的东西。 这可能是一个问题，也可能不是 - 这取决于我们试图使用连接表执行的分析。

湾区共享单车

在本章结尾，我们通过使用我们学过的所有方法，来检验新的大型数据集。 我们还将介绍一个强大的可视化工具`map_table`。

湾区自行车共享服务公司在其系统中发布了一个数据集，描述了 2014 年 9 月到 2015 年 8 月期间的每个自行车的租赁。 总共有 354152 次出租。 表的列是：

- 租赁 ID
- 租赁的时间，以秒为单位
- 开始日期
- 起点站的名称和起始终端的代码
- 终点站的名称和终止终端的代码
- 自行车的序列号
- 订阅者类型和邮政编码

```
trips = Table.read_table('trip.csv')
trips
```

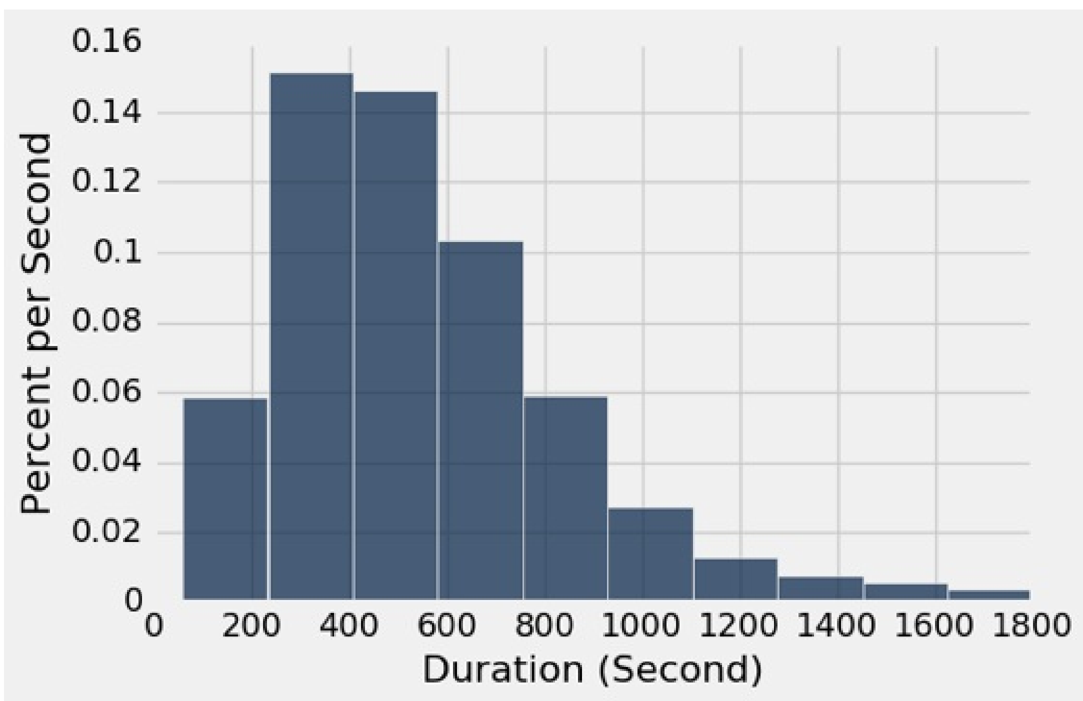
Trip ID	Duration	Start Date	Start Station	Start Terminal	End Date	End St
913460	765	8/31/2015 23:26	Harry Bridges Plaza (Ferry Building)	50	8/31/2015 23:39	San Francis Caltrain (Towns 4th)
913459	1036	8/31/2015 23:11	San Antonio Shopping Center	31	8/31/2015 23:28	Mounta View Ci Hall
913455	307	8/31/2015 23:13	Post at Kearny	47	8/31/2015 23:18	2nd at S Park
913454	409	8/31/2015 23:10	San Jose City Hall	10	8/31/2015 23:17	San Sa at 1st
913453	789	8/31/2015 23:09	Embarcadero at Folsom	51	8/31/2015 23:22	Embarc at Sans
913452	293	8/31/2015 23:07	Yerba Buena Center of the Arts (3rd @ Howard)	68	8/31/2015 23:12	San Francis Caltrain (Towns 4th)
913451	896	8/31/2015 23:07	Embarcadero at Folsom	51	8/31/2015 23:22	Embarc at Sans
913450	255	8/31/2015 22:16	Embarcadero at Sansome	60	8/31/2015 22:20	Steuart Market
913449	126	8/31/2015 22:12	Beale at Market	56	8/31/2015 22:15	Tempor Transba Termina (Howar Beale)
913448	932	8/31/2015 21:57	Post at Kearny	47	8/31/2015 22:12	South V Ness at Market

（省略了 354142 行）

我们只专注于免费行程，这是持续不到 1800 秒（半小时）的行程。 长途行程需要付费。

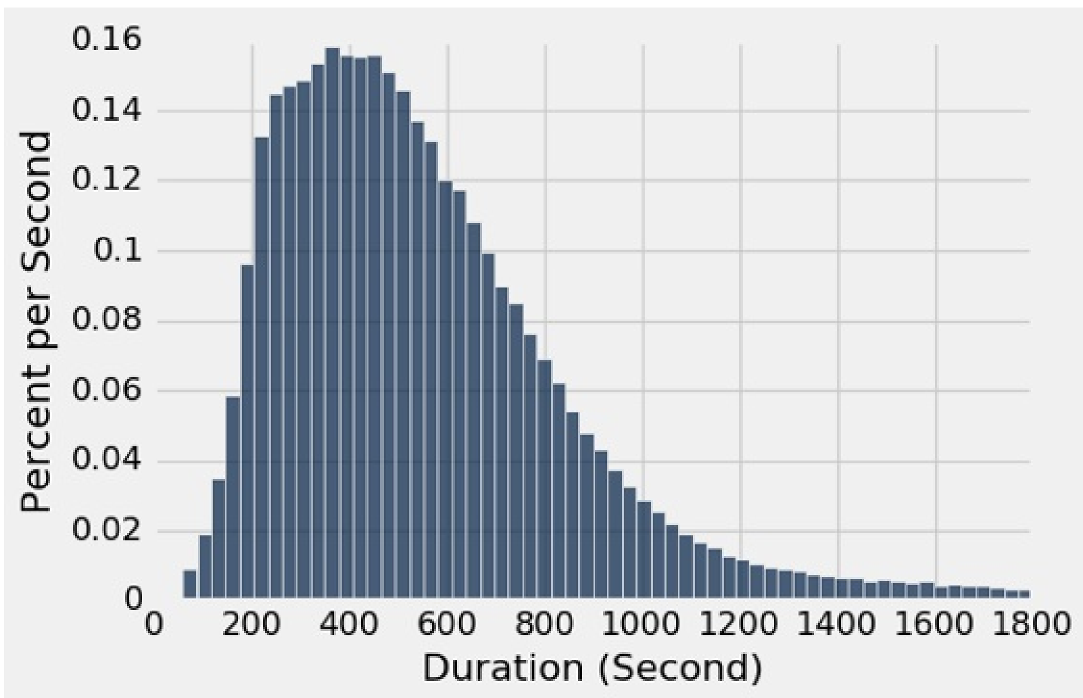
下面的直方图显示，大部分行程需要大约 10 分钟（600 秒）左右。 很少有人花了近 30 分钟（1800 秒），可能是因为人们试图在截止时间之前退还自行车，以免付费。

```
commute = trips.where('Duration', are.below(1800))
commute.hist('Duration', unit='Second')
```



我们可以通过指定更多的桶来获得更多的细节。 但整体形状并没有太大变化。

```
commute.hist('Duration', bins=60, unit='Second')
```



使用 `group` 和 `pivot` 探索数据

我们可以使用 `group` 来识别最常用的起点站。

```
starts = commute.group('Start Station').sort('count', descending=True)
starts
```

Start Station	count
San Francisco Caltrain (Townsend at 4th)	25858
San Francisco Caltrain 2 (330 Townsend)	21523
Harry Bridges Plaza (Ferry Building)	15543
Temporary Transbay Terminal (Howard at Beale)	14298
2nd at Townsend	13674
Townsend at 7th	13579
Steuart at Market	13215
Embarcadero at Sansome	12842
Market at 10th	11523
Market at Sansome	11023

（省略了 60 行）

大多数行程起始于 Townsend 的 Caltrain 站，和旧金山的四号车站。 人们乘坐火车进入城市，然后使用共享单车到达下一个目的地。

`group` 方法也可以用于按照起点站和终点站，对租赁进行分类。

```
commute.group(['Start Station', 'End Station'])
```

Start Station	End Station	count
2nd at Folsom	2nd at Folsom	54
2nd at Folsom	2nd at South Park	295
2nd at Folsom	2nd at Townsend	437
2nd at Folsom	5th at Howard	113
2nd at Folsom	Beale at Market	127
2nd at Folsom	Broadway St at Battery St	67
2nd at Folsom	Civic Center BART (7th at Market)	47
2nd at Folsom	Clay at Battery	240
2nd at Folsom	Commercial at Montgomery	128
2nd at Folsom	Davis at Jackson	28

（省略了 1619 行）

共有五十四次行程开始和结束于在 Folsom 二号车站，。 有很多人（437 人）往返于 Folsom 二号和 Townsend 二号车站之间。

`pivot` 方法执行相同的分类，但将结果显示在一个透视表中，该表显示了起点和终点站的所有可能组合，即使其中一些不对应任何行程。 请记住，`pivot` 函数的第一个参数指定了数据透视表的列标签；第二个参数指定行标签。

在 Beale at Market 附近有一个火车站以及一个湾区快速公交（BART）站，解释了从那里开始和结束的大量行程。

```
commute.pivot('Start Station', 'End Station')
```

End Station	2nd at Folsom	2nd at South Park	2nd at Townsend	5th at Howard	Adobe on Almaden	Arena Green / SAP Center	Be a Ma
2nd at Folsom	54	190	554	107	0	0	40
2nd at South Park	295	164	71	180	0	0	208
2nd at Townsend	437	151	185	92	0	0	608
5th at Howard	113	177	148	83	0	0	59
Adobe on Almaden	0	0	0	0	11	4	0
Arena Green / SAP Center	0	0	0	0	7	64	0
Beale at Market	127	79	183	59	0	0	59
Broadway St at Battery St	67	89	279	119	0	0	102
California Ave Caltrain Station	0	0	0	0	0	0	0
Castro Street and El Camino Real	0	0	0	0	0	0	0

（省略了 60 行）

我们也可以使用 `pivot` 来寻找起点和终点站之间的最短骑行时间。在这里，`pivot` 已经接受了可选参数 `Duration`，以及 `min` 函数，它对每个单元格中的值执行。

```
commute.pivot('Start Station', 'End Station', 'Duration', min)
```

End Station	2nd at Folsom	2nd at South Park	2nd at Townsend	5th at Howard	Adobe on Almaden	Arena Green / SAP Center	Be a Ma
2nd at Folsom	61	97	164	268	0	0	27
2nd at South Park	61	60	77	86	0	0	78
2nd at Townsend	137	67	60	423	0	0	31
5th at Howard	215	300	384	68	0	0	35
Adobe on Almaden	0	0	0	0	84	275	0
Arena Green / SAP Center	0	0	0	0	305	62	0
Beale at Market	219	343	417	387	0	0	60
Broadway St at Battery St	351	424	499	555	0	0	19
California Ave Caltrain Station	0	0	0	0	0	0	0
Castro Street and El Camino Real	0	0	0	0	0	0	0

（省略了 60 行）

有人从 Folsom 的二号车站骑到 Beale at Market，距离大约五个街区，非常快（271 秒，约 4.5 分钟）。2nd Avenue 和 Almaden 的 Adobe 之间没有自行车行程，因为后者在不同的城市。

绘制地图

`stations` 表包含每个自行车站的地理信息，包括纬度，经度和“地标”，它是该站所在城市的名称。

```
stations = Table.read_table('station.csv')
stations
```

station_id	name	lat	long	dockcount	landmark	installa
2	San Jose Diridon Caltrain Station	37.3297	-121.902	27	San Jose	8/6/201
3	San Jose Civic Center	37.3307	-121.889	15	San Jose	8/5/201
4	Santa Clara at Almaden	37.334	-121.895	11	San Jose	8/6/201
5	Adobe on Almaden	37.3314	-121.893	19	San Jose	8/5/201
6	San Pedro Square	37.3367	-121.894	15	San Jose	8/7/201
7	Paseo de San Antonio	37.3338	-121.887	15	San Jose	8/7/201
8	San Salvador at 1st	37.3302	-121.886	15	San Jose	8/5/201
9	Japantown	37.3487	-121.895	15	San Jose	8/5/201
10	San Jose City Hall	37.3374	-121.887	15	San Jose	8/6/201
11	MLK Library	37.3359	-121.886	19	San Jose	8/6/201

（省略了 60 行）

我们可以使用 `Marker.map_table` 来绘制一个车站所在位置的地图。 该函数在一个表格上进行操作，该表格的列依次是纬度，经度以及每个点的可选标识符。

```
Marker.map_table(stations.select('lat', 'long', 'name'))
```

地图 1

地图使用 OpenStreetMap 创建的，OpenStreetMap 是一个开放的在线地图系统，你可以像使用 Google 地图或任何其他在线地图一样使用。 放大到旧金山，看看车站如何分布。 点击一个标记，看看它是哪个站。

你也可以用彩色圆圈表示地图上的点。 这是旧金山自行车站的地图。

```
sf = stations.where('landmark', are.equal_to('San Francisco'))
sf_map_data = sf.select('lat', 'long', 'name')
Circle.map_table(sf_map_data, color='green', radius=200)
```

地图 2

更多信息的地图：`join` 的应用

自行车站位于湾区五个不同的城市。 为了区分每个城市，通过使用不同的颜色，我们首先使用 `group` 来标识所有城市，并为每个城市分配一个颜色。

```
cities = stations.group('landmark').relabeled('landmark', 'city')
cities
```

city	count
Mountain View	7
Palo Alto	5
Redwood City	7
San Francisco	35
San Jose	16

```
colors = cities.with_column('color', make_array('blue', 'red', 'green', 'orange', 'pur
ple'))
colors
```

city	count	color
Mountain View	7	blue
Palo Alto	5	red
Redwood City	7	green
San Francisco	35	orange
San Jose	16	purple

现在我们可以按照 `landmark` 连接 `stations` 和 `colors` ，之后选取绘制地图所需的列。

```
joined = stations.join('landmark', colors, 'city')
colored = joined.select('lat', 'long', 'name', 'color')
Marker.map_table(colored)
```

地图 3

现在五个不同城市由五种不同颜色标记。

要查看大部分自行车租赁的来源，让我们确定起点站：

```
starts = commute.group('Start Station').sort('count', descending=True)
starts
```

Start Station	count
San Francisco Caltrain (Townsend at 4th)	25858
San Francisco Caltrain 2 (330 Townsend)	21523
Harry Bridges Plaza (Ferry Building)	15543
Temporary Transbay Terminal (Howard at Beale)	14298
2nd at Townsend	13674
Townsend at 7th	13579
Steuart at Market	13215
Embarcadero at Sansome	12842
Market at 10th	11523
Market at Sansome	11023

（省略了 60 行）

我们可以包含映射这些车站所需的地理数据，首先连接 `starts` 的 `stations`：

```
station_starts = stations.join('name', starts, 'Start Station')
station_starts
```

name	station_id	lat	long	dockcount	landmark	installa
2nd at Folsom	62	37.7853	-122.396	19	San Francisco	8/22/20
2nd at South Park	64	37.7823	-122.393	15	San Francisco	8/22/20
2nd at Townsend	61	37.7805	-122.39	27	San Francisco	8/22/20
5th at Howard	57	37.7818	-122.405	15	San Francisco	8/21/20
Adobe on Almaden	5	37.3314	-121.893	19	San Jose	8/5/201
Arena Green / SAP Center	14	37.3327	-121.9	19	San Jose	8/5/201
Beale at Market	56	37.7923	-122.397	19	San Francisco	8/20/20
Broadway St at Battery St	82	37.7985	-122.401	15	San Francisco	1/22/20
California Ave Caltrain Station	36	37.4291	-122.143	15	Palo Alto	8/14/20
Castro Street and El Camino Real	32	37.386	-122.084	11	Mountain View	12/31/2

（省略了 58 行）

现在我们只提取绘制地图所需的数据，为每个站添加一个颜色和一个面积。 面积是起始于每个站点的租用次数的 1000 倍，其中选择了常数 1000，以便在地图上以适当的比例绘制圆圈。

```
starts_map_data = station_starts.select('lat', 'long', 'name').with_columns(
    'color', 'blue',
    'area', station_starts.column('count') * 1000
)
starts_map_data.show(3)
Circle.map_table(starts_map_data)
```

lat	long	name	color	area
37.7853	-122.396	2nd at Folsom	blue	7841000
37.7823	-122.393	2nd at South Park	blue	9274000
37.7805	-122.39	2nd at Townsend	blue	13674000

（省略了 65 行）

地图 4

旧金山的一大块表明，这个城市的东部是湾区自行车租赁的重点区域。

八、随机性

> 原文：Randomness
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在前面的章节中，我们开发了深入描述数据所需的技能。 数据科学家也必须能够理解随机性。 例如，他们必须能够随机将个体分配到实验组和对照组，然后试图说明，观察到的两组结果之间的差异是否仅仅是由于随机分配，或真正由于实验所致。

在这一章中，我们开始分析随机性。 首先，我们将使用 Python 进行随机选择。 在 `numpy` 中有一个叫做 `random` 的子模块，它包含许多涉及随机选择的函数。 其中一个函数称为 `choice` 。 它从一个数组中随机选取一个项目，选择任何项目都是等可能的。 函数调用是 `np.random.choice(array_name)` ，其中 `array_name` 是要从中进行选择的数组的名称。

因此，下面的代码以 50% 的几率求值为 `treatment` ，50% 的机率为 `control` 。

```
two_groups = make_array('treatment', 'control')
np.random.choice(two_groups)
'treatment'
```

上面的代码和我们迄今运行的所有其他代码之间的巨大差异在于，上面的代码并不总是返回相同的值。 它可以返回 `treatment` 或 `control` ，我们不会提前知道会选择哪一个。 我们可以通过提供第二个参数来重复这个过程，它是重复这个过程的次数。

```
np.random.choice(two_groups, 10)
array(['treatment', 'control', 'treatment', 'control', 'control',
       'treatment', 'treatment', 'control', 'control', 'control'],
      dtype='<U9')
```

随机事件的根本问题是它们是否发生。 例如：

- 个体是否被分配到实验组？
- 赌徒是否会赢钱？
- 一个民意调查是否做出了准确的预测？

一旦事件发生，你可以对所有这些问题回答“是”或“否”。 在编程中，通常通过将语句标记为 `True` 或 `False` 来执行此操作。 例如，如果个体被分配到实验组，那么“个体被分配到实验组”的陈述将是真的。 如果不是，那将是假的。

布尔值和比较

在 Python 中，布尔值（以逻辑学家 George Boole 命名）表示真值，并只有两个可能的值：`True` 和 `False` 。 无论问题是否涉及随机性，布尔值通常都由比较运算符产生。 Python 包含了各种比较值的运算符。 例如，`3` 大于 `1 + 1` 。

```
3 > 1 + 1
True
```

`True` 表示比较是有效的；Python 已经证实了 `3` 和 `1 + 1` 的关系的这个简单事实。 下面列出了一整套通用的比较运算符。

比较	运算符	True 示例	False 示例
小于	`<`	`2 < 3`	`2 < 2`
大于	`>`	`3 > 2`	`3 > 3`
小于等于	`<=`	`2 <= 2`	`3 <= 2`
大于等于	`>=`	`3 >= 3`	`2 >= 3`
等于	`==`	`3 == 3`	`3 == 2`
不等于	`!=`	`3 != 2`	`2 != 2`

注意比较中的两个等号 `==` 用于确定相等性。 这是必要的，因为 Python 已经使用 `=` 来表示名称的赋值，我们之前看到过。 它不能将相同的符号用于不同的目的。 因此，如果你想检查 `5` 是否等于 `10/2` ，那么你必须小心： `5 = 10/` 2返回一个错误信息，因为 Python 假设你正试图将表达式 `10/2` 的值赋给一个名称，它是数字 `5` 。相反，你必须使用 `5 == 10/2` ，其计算结果为 `True` 。

```
5 = 10/2
  File "<ipython-input-4-5c7d3e808777>", line 1
    5 = 10/2
            ^
SyntaxError: can't assign to literal
5 == 10/2
True
```

一个表达式可以包含多个比较，并且它们都必须满足，为了整个表达式为真。 例如，我们可以用下面的表达式表示 `1 + 1` 在 `1` 和 `3` 之间。

```
1 < 1 + 1 < 3
True
```

两个数字的平均值总是在较小的数字和较大的数字之间。 我们用下面的数字 `x` 和 `y` 来表示这种关系。 你可以尝试不同的 `x` 和 `y` 值来确认这种关系。

```
x = 12
y = 5
min(x, y) <= (x+y)/2 <= max(x, y)
True
```

字符串比较

字符串也可以比较，他们的顺序是字典序。 较短的字符串小于以较短的字符串开头的较长的字符串。

```
'Dog' > 'Catastrophe' > 'Cat'
```

我们回到随机选择。 回想一下由两个元素组成的数组 `two_groups` ， `treatment` 和 `control` 。为了看一个随机分配的个体是否去了实验组，你可以使用比较：

```
np.random.choice(two_groups) == 'treatment'
False
```

和以前一样，随机选择并不总是一样的，所以比较的结果也不总是一样的。 这取决于是选择 `treatment` 还是 `control` 。 对于任何涉及随机选择的单元格，多次运行单元格来获得结果的变化是一个好主意。

比较数组和值

回想一下，我们可以对数组中的很多数字执行算术运算。 例如， `make_array(0, 5, 2)*2` 等同于 `make_array(0, 10, 4)` 。 以类似的方式，如果我们比较一个数组和一个值，则数组的每个元素都与该值进行比较，并将比较结果求值为布尔值数组。

```
tosses = make_array('Tails', 'Heads', 'Tails', 'Heads', 'Heads')
tosses == 'Heads'
array([False,  True, False,  True,  True], dtype=bool)
```

`numpy` 方法 `count_nonzero` 计算数组的非零（即 `True` ）元素的数量。

```
np.count_nonzero(tosses == 'Heads')
3
```

条件语句

在许多情况下，行动和结果取决于所满足的一组特定条件。例如，随机对照试验的个体如果被分配给实验组，则接受实验。赌徒如果赢了赌注就赚钱。

在本节中，我们将学习如何使用代码来描述这种情况。条件语句是一个多行语句，它允许 Python 根据表达式的真值选择不同的选项。虽然条件语句可以出现在任何地方，但它们通常出现在函数体内，以便根据参数值执行可变的行为。

条件语句总是以 `if` 开头，这是一行，后面跟着一个缩进的主体。只有当 `if` 后面的表达式（称为 `if` 表达式）求值为真时，才会执行主体。如果 `if` 表达式的计算结果为 `False`，则跳过 `if` 的主体。

让我们开始定义一个返回数字符号的函数。

```
def sign(x):

    if x > 0:
        return 'Positive'
sign(3)
'Positive'
```

如果输入是正数，则此函数返回正确的符号。但是，如果输入不是正数，那么 `if` 表达式的计算结果为 `false`，所以 `return` 语句被跳过，函数调用没有值（为 `None`）。

```
sign(-3)
```

所以，让我们改进我们的函数来返回负数，如果输入是负数。我们可以通过添加一个 `elif` 子句来实现，其中 `elif` 是 Python 的 `else, if` 的缩写。

```
def sign(x):

    if x > 0:
        return 'Positive'

    elif x < 0:
        return 'Negative'
```

现在当输入为 `-3` 时，`sign` 返回正确答案。

```
sign(-3)
'Negative'
```

那么如果输入是 `0` 呢？为了处理这个情况，我们可以添加 `elif` 子句：

```
def sign(x):

    if x > 0:
        return 'Positive'

    elif x < 0:
        return 'Negative'

    elif x == 0:
        return 'Neither positive nor negative'
sign(0)
'Neither positive nor negative'
```

与之等价，我们可以用 `else` 子句替换最后的 `elif` 子句，只有前面的所有比较都是 `false`，才会执行它的正文。 也就是说，输入值等于 `0` 的时候。

```
def sign(x):

    if x > 0:
        return 'Positive'

    elif x < 0:
        return 'Negative'

    else:
        return 'Neither positive nor negative'
sign(0)
'Neither positive nor negative'
```

一般形式

条件语句也可以有多个具有多个主体的子句，只有其中一个主体可以被执行。 多子句的条件语句的一般格式如下所示。

```
if <if expression>:
    <if body>
elif <elif expression 0>:
    <elif body 0>
elif <elif expression 1>:
    <elif body 1>
...
else:
    <else body>
```

总是只有一个 `if` 子句，但是可以有任意数量的 `elif` 子句。 Python 将依次求解头部的 `if` 和 `elif` 表达式，直到找到一个真值，然后执行相应的主体。 `else` 子句是可选的。 当提供 `else` 头部时，只有在前面的子句的头部表达式都不为真时才执行 `else` 头部。 `else` 子句必须总是在最后（或根本没有）。

示例："另一个"

现在我们将使用条件语句来定义一个看似相当虚假和对立的函数，但是在本章后面的章节中会变得方便。 它需要一个数组，包含两个元素（例如， `red` 和 `blue` ），以及另一个用于比较的元素。 如果该元素为 `red` ，则该函数返回 `blue` 。 如果元素是（例如） `blue` ，则函数返回 `red` 。 这就是为什么我们要将函数称为 `other_one` 。

```
def other_one(x, a_b):

    """Compare x with the two elements of a_b;
    if it is equal to one of them, return the other one;
    if it is not equal to either of them, return an error message.
    """
    if x == a_b.item(0):
        return a_b.item(1)

    elif x == a_b.item(1):
        return a_b.item(0)

    else:
        return 'The input is not valid.'
colors = make_array('red', 'blue')
other_one('red', colors)
'blue'
other_one('blue', colors)
'red'
other_one('potato', colors)
'The input is not valid.'
```

迭代

编程中经常出现这样的情况，特别是在处理随机性时，我们希望多次重复一个过程。 例如，要检查 `np.random.choice` 是否实际上是随机选取的，我们可能需要多次运行下面的单元格，以查看 `Heads` 是否以大约 50% 的几率出现。

```
np.random.choice(make_array('Heads', 'Tails'))
'Heads'
```

我们可能希望重新运行代码，带有稍微不同的输入或其他稍微不同的行为。 我们可以多次复制粘贴代码，但是这很枯燥，容易出现拼写错误，如果我们想要这样做一千次或一百万次，忘记它吧。

更自动化的解决方案是使用 `for` 语句遍历序列的内容。 这被称为迭代。 `for` 语句以单词 `for` 开头，后面跟着一个名字，我们要把这个序列中的每个项目赋给它，后面跟着单词 `in` ，最后以一个表达式结束，它求值为一个序列。 对于序列中的每个项目， `for` 语句的缩进主体执行一次。

```
for i in np.arange(3):
    print(i)
0
1
2
```

想象一下，没有 `for` 语句的情况下，完全实现 `for` 语句功能的代码，这样很有帮助。（这被称为循环展开。） `for` 语句简单地复制了内部的代码，但是在每次迭代之前，它从给定的序列中将我们选择的名称赋为一个新的值。 例如，以下是上面循环的展开版本：

```
i = np.arange(3).item(0)
print(i)
i = np.arange(3).item(1)
print(i)
i = np.arange(3).item(2)
print(i)
0
1
2
```

> 译者注：实际的实现方式不是这样，但是效果一样。这里不做深究。

请注意，我的名字是任意的，就像我们用 `=` 赋值的名字一样。

在这里我们用一个更为现实的方式使用 `for` 语句：我们从数组中打印 `5` 个随机选项。

```
coin = make_array('Heads', 'Tails')

for i in np.arange(5):
    print(np.random.choice(make_array('Heads', 'Tails')))
Heads
Heads
Tails
Heads
Heads
```

在这种情况下，我们只执行了几次完全相同的（随机）操作，所以我们 `for` 语句中的代码实际上并不涉及到 `i` 。

扩展数组

虽然上面的 `for` 语句确实模拟了五次硬币投掷的结果，但结果只是简单地打印出来，并不是我们可以用来计算的形式。 因此， `for` 语句的典型用法是创建一个结果数组，每次都扩展它。

`numpy` 中的 `append` 方法可以帮助我们实现它。 调用 `np.append(array_name，value)` 将求出一个新的数组，它是由 `value` 扩展的 `array_name` 。在使用 `append` 时请记住，数组的所有条目必须具有相同的类型。

```
pets = make_array('Cat', 'Dog')
np.append(pets, 'Another Pet')
array(['Cat', 'Dog', 'Another Pet'],
      dtype='<U11')
```

这会使 `pets` 数组保持不变。

```
pets
array(['Cat', 'Dog'],
      dtype='<U3')
```

但是在扩展数组的时候，通常使用 `for` 循环来修改它很方便。 这通过将扩展后的数组赋给原始数组的相同名称来实现。

```
pets = np.append(pets, 'Another Pet')
pets
array(['Cat', 'Dog', 'Another Pet'],
      dtype='<U11')
```

示例：计算正面的数量

现在我们可以模拟一个硬币的五次投掷，并把结果放入一个数组中。 我们将从创建一个空数组开始，然后附加每次投掷的结果。

```
coin = make_array('Heads', 'Tails')

tosses = make_array()

for i in np.arange(5):
    tosses = np.append(tosses, np.random.choice(coin))

tosses
array(['Tails', 'Heads', 'Tails', 'Heads', 'Tails'],
      dtype='<U32')
```

让我们将 `for` 语句展开，重写单元格。

```
coin = make_array('Heads', 'Tails')

tosses = make_array()

i = np.arange(5).item(0)
tosses = np.append(tosses, np.random.choice(coin))
i = np.arange(5).item(1)
tosses = np.append(tosses, np.random.choice(coin))
i = np.arange(5).item(2)
tosses = np.append(tosses, np.random.choice(coin))
i = np.arange(5).item(3)
tosses = np.append(tosses, np.random.choice(coin))
i = np.arange(5).item(4)
tosses = np.append(tosses, np.random.choice(coin))

tosses
array(['Heads', 'Heads', 'Tails', 'Tails', 'Heads'],
      dtype='<U32')
```

通过将结果捕获到数组中，我们自己有能力使用数组方法进行计算。 例如，我们可以使用 `np.count_nonzero` 来计算五次投掷中的正面数量。

```
np.count_nonzero(tosses == 'Heads')
2
```

迭代是一个强大的技术。 例如，通过为 1000 次投掷运行完全相同的代码，而不是 `5` 次，我们可以计算 `1000` 次投掷的正面数量。

```
tosses = make_array()

for i in np.arange(1000):
    tosses = np.append(tosses, np.random.choice(coin))

np.count_nonzero(tosses == 'Heads')
481
```

示例：100 次投掷中的正面数量

预测 100 次硬币投掷中有 50 个正面是很自然的，或多或少。

但多少是“或多或少”呢？ 获得正好 50 个正面的几率是多少？ 像数据科学这样的问题，不仅因为它们涉及随机性的有趣方面，而且因为它们可以用于分析试验，其中实验和控制组的分配由硬币的投掷决定。

在这个例子中，我们将模拟以下实验的 10,000 次重复：

- 掷硬币 100 次，记录正面数量。

我们的结果的直方图会让我们了解有多少个正面。

作为一个预热，请注意，`np.random.choice` 接受可选的第二个参数来指定选择的数量。 默认情况下，选择使用替换来进行。 这里是一个硬币 10 次投掷的模拟：

```
np.random.choice(coin, 10)
array(['Tails', 'Heads', 'Heads', 'Tails', 'Tails', 'Heads', 'Tails',
       'Tails', 'Heads', 'Tails'],
      dtype='<U5')
```

现在我们来研究 100 次投掷。 我们将首先创建一个名为 `heads` 的空数组。 然后，在每个一万次重复中，我们会抛硬币 100 次，计算正面的数量，并将其附加到 `heads` 上。

```
N = 10000

heads = make_array()

for i in np.arange(N):
    tosses = np.random.choice(coin, 100)
    heads = np.append(heads, np.count_nonzero(tosses == 'Heads'))

heads
array([ 46.,  64.,  59., ...,  56.,  54.,  56.])
```

让我们将结果收集到表格中，并绘制直方图：

```
results = Table().with_columns(
    'Repetition', np.arange(1, N+1),
    'Number of Heads', heads
)

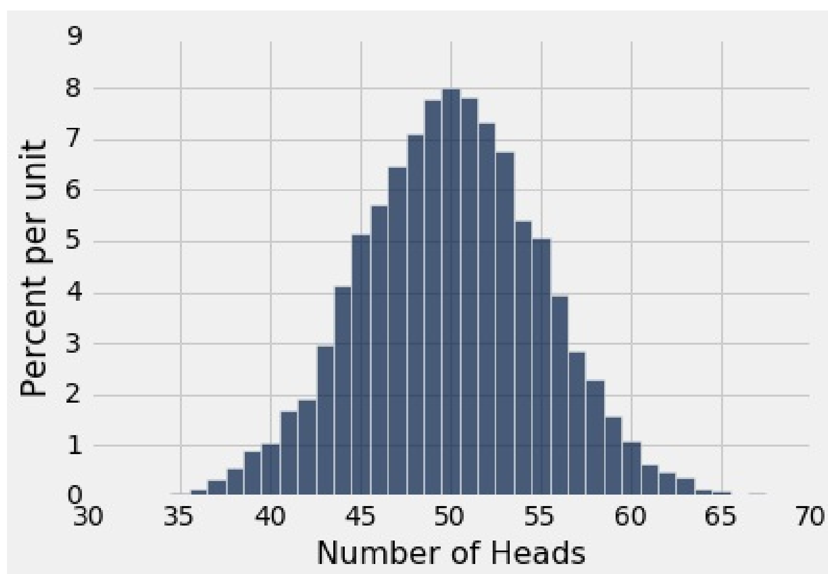
results
```

Repetition	Number of Heads
1	46
2	64
3	59
4	57
5	54
6	47
7	45
8	50
9	44
10	57

（省略了 9990 行）

这里是数据的直方图，桶的宽度为 1，中心为每个正面数量的值。

```
results.select('Number of Heads').hist(bins=np.arange(30.5, 69.6, 1))
```



毫不奇怪，直方图看起来大约关于 50 个正面左右对称。 50 处的条形的高度大约是每单位 8%。 由于每个条形的宽度都是 1 个单位，这就是说，8% 的重复正好产生了 50 个正面。 这不是一个很大的百分比，但是与其他数量的正面相比，这是最大的。

直方图还显示，在几乎所有的重复中，100 次投掷的正面数量在 35 到 65 之间。事实上，大部分的重复产生 45 到 55 个正面数量。

理论上，正面数量可能在 0 到 100 之间，但模拟显示可能值的范围要小得多。

这是一个更普遍现象的例子，关于掷硬币中的变化，我们将在后面看到。

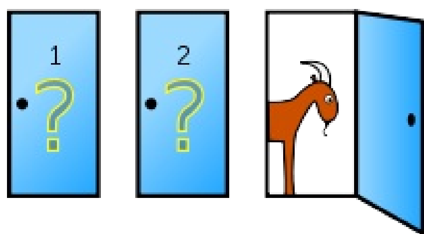
Monty Hall 问题

多年来这个问题已经使许多人感到困惑，包括数学家在内。 让我们看看我们是否可以解决。

这个设定来源于一个名为“让我们做个交易”（Let's Make a Deal）的电视游戏节目。Monty Hall 在二十世纪六十年代主持了这个节目，从此产生了一些副产品。 这个节目令人兴奋的一部分是，虽然参赛者有机会赢得大奖，但他们可能最终会选择不那么理想的“zonks”。 这就是现在所谓的 Monty Hall 问题的基础。

这个设定是一个游戏节目，参赛者面对三个闭着的门。 在其中一扇门的后面是一辆奇特的汽车，另外两扇门后面有一只山羊。 参赛者不知道汽车的位置，必须按照以下规则进行尝试。

- 参赛者进行初步选择，但不打开那个门。
- 其他两个门中至少有一个门的后面必须有一只山羊。Monty 打开这些门之一来展示山羊，维基百科中显示了他所有的荣耀。



- 还剩下两个门，其中一个是参赛者的原始选择。 其中一扇门后面有车，另一扇有一只山羊。 参赛者现在可以选择打开两扇门中的哪一扇。

参赛者需要作出决定。 如果她想要这辆车，她应该选择打开哪扇门？ 她应该坚持最初的选择，还是转向另一个门？ 这是 Monty Hall 问题。

解法

在涉及几率的任何问题中，重要的随机性的假设。 假设有三分之一的几率，参赛者的最初选择是后面有车的门，这是合理的。

在这个假设下，解决这个问题的方法非常简单，尽管简单的解决方案并不能说服每个人。 无论如何就是这样。

- 汽车在原来选择的门后面的几率是 1/3。
- 汽车在原来选择的门后面或者剩余的门后面。 它不能在其他地方。
- 因此，汽车在剩余的门后的几率是 2/3。
- 因此，选手应该更改选择。
- 就是这样，故事结束了。

不相信？ 那么让我们模拟游戏，看看结果如何。

模拟

我们开始建立两个实用的数组，`doors`和`goats`，这会让我们区分三个门和两只山羊。

```
doors = make_array('Car', 'Goat 1', 'Goat 2')
goats = make_array('Goat 1', 'Goat 2')
```

现在我们定义一个函数`monty_hall`来模拟游戏，并按照这个顺序返回含有三个字符串的数组：

- 参赛选手的原始选择的什么
- Monty 排除了什么
- 剩下的门是什么

如果选手的原始选择是带山羊的门，蒙蒂必须扔掉另一只山羊，剩下的就是这辆车。 如果最初的选择是带车的门，蒙蒂必须扔掉两只山羊中的一只，剩下的就是另一只羊。

因此很显然，在前一节中定义的函数将是有用的。 它需要一个字符串和一个两个元素的数组;如果字符串等于其中一个元素，则返回另一个元素。

```
def other_one(x, a_b):
    if x == a_b.item(0):
        return a_b.item(1)
    elif x == a_b.item(1):
        return a_b.item(0)
    else:
        return 'Input Not Valid'
```

如果选手的原始选择是山羊，游戏的结果是这二者之一：

```
original = 'Goat 1'
make_array(original, other_one(original, goats), 'Car')
array(['Goat 1', 'Goat 2', 'Car'],
      dtype='<U6')
original = 'Goat 2'
make_array(original, other_one(original, goats), 'Car')
array(['Goat 2', 'Goat 1', 'Car'],
      dtype='<U6')
```

现在我们可以把所有这些代码放到`monty_hall`函数中，来模拟一次游戏的结果。 该函数不带任何参数。

参赛者的原始选择将是三门之中随机选择的门。

为了检查原始选择是否是山羊，我们首先写一个名为`is_goat`的小函数：

```
def is_goat(door_name):
    """ Check whether the name of a door (a string) is a Goat.

    Examples:
    =========

    >>> is_goat('Goat 1')
    True
    >>> is_goat('Goat 2')
    True
    >>> is_goat('Car')
    False
    """
    if door_name == "Goat 1":
        return True
    elif door_name == "Goat 2":
        return True
    else:
        return False


def monty_hall():

    """ Play the Monty Hall game once
    and return an array of three strings:

    original choice, what Monty throws out, what remains
    """

    original = np.random.choice(doors)

    if is_goat(original):
        return make_array(original, other_one(original, goats), 'Car')

    else:
        throw_out = np.random.choice(goats)
        return make_array(original, throw_out, other_one(throw_out, goats))
```

让我们玩几次这个游戏。这里是一个结果。你应该运行几次单元格来观察结果如何变化。

```
monty_hall()
array(['Car', 'Goat 2', 'Goat 1'],
      dtype='<U6')
```

为了衡量不同结果发生的频率，我们必须玩多次游戏并收集结果。 为此，我们将使用`for`循环。

我们将首先定义三个空数组，每个数组对应原始选择，Monty 排除了什么，剩下的是什么。然后我们将玩这个游戏 N 次并收集结果。我们已经将 N 设为 10,000，但是你可以改变它。

```
# Number of times we'll play the game
N = 10000

original = make_array()      # original choice
throw_out = make_array()     # what Monty throws out
remains = make_array()       # what remains

for i in np.arange(N):
    result = monty_hall()     # the result of one game

    # Collect the results in the appropriate arrays
    original = np.append(original, result.item(0))
    throw_out = np.append(throw_out, result.item(1))
    remains = np.append(remains, result.item(2))

# The for-loop is done! Now put all the arrays together in a table.
results = Table().with_columns(
    'Original Door Choice', original,
    'Monty Throws Out', throw_out,
    'Remaining Door', remains
)
results
```

Original Door Choice	Monty Throws Out	Remaining Door
Car	Goat 1	Goat 2
Goat 1	Goat 2	Car
Goat 2	Goat 1	Car
Car	Goat 2	Goat 1
Car	Goat 2	Goat 1
Goat 1	Goat 2	Car
Goat 1	Goat 2	Car
Goat 1	Goat 2	Car
Goat 2	Goat 1	Car
Goat 1	Goat 2	Car

（省略了 9990 行）

为了看看选手是否应该坚持原来的选择或更改，让我们看看她的两个选项后面的车的频率。

```
results.group('Original Door Choice')
```

Original Door Choice	count
Car	3312
Goat 1	3382
Goat 2	3306

```
results.group('Remaining Door')
```

Remaining Door	count
Car	6688
Goat 1	1640
Goat 2	1672

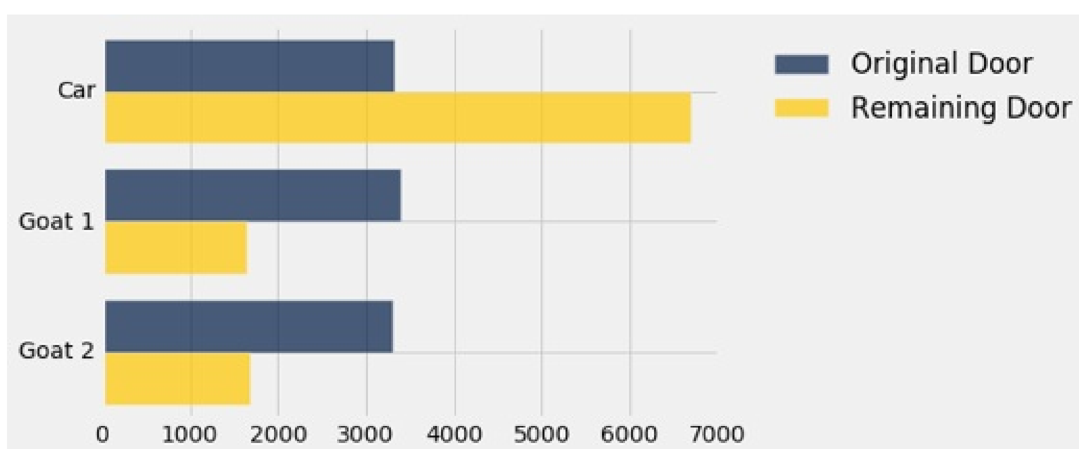
我们的解决方案说明了，这辆车有三分之二的几率在剩下的门后面，这是相当不错的近似值。 如果参赛者更改了她的选择，她有两倍的可能性会得到车。

为了使结果可视化，我们可以将上面的两个表格连接在一起并绘制叠加的条形图。

```
results_o = results.group('Original Door Choice')
results_r = results.group('Remaining Door')
joined = results_o.join('Original Door Choice', results_r, 'Remaining Door')
combined = joined.relabeled(0, 'Item').relabeled(1, 'Original Door').relabeled(2, 'Rem
aining Door')
combined
```

Item	Original Door	Remaining Door
Car	3312	6688
Goat 1	3382	1640
Goat 2	3306	1672

```
combined.barh(0)
```



注意三条蓝色条形几乎相等 - 原始选择有同等可能是三个可用条目中的任何一条。 但是，汽车对应的金色条形是蓝色条形的两倍。

模拟证实了，如果参赛者改变选择，她有两倍的可能性获胜。

发现概率

几个世纪以来，对于什么是概率存在哲学争论。有些人认为概率是相对频率；其他人认为他们是长期的相对频率较长；还有一些人认为概率是个人不确定性程度的主观测量。

在这个课程中，大多数概率将是相对频率，尽管许多人会有主观的解释。无论如何，在不同的解释中，概率计算和组合的方式是一致的。

按照惯例，概率是介于 0 和 1 之间的数字，或者 0% 和 100% 之间。不可能的事件概率为 0。确定的事件概率为 1。

数学是准确发现概率的主要工具，尽管计算机也可用于此目的。模拟可以提供出色的近似，具有很高的概率。在本节中，我们将以非正式方式制定一些简单的规则来管理概率的计算。在随后的章节中，我们将回到模拟来近似复杂事件的概率。

我们将使用标准符号 $P(\text{event})$ 来表示“事件”发生的概率，我们将交替使用“几率”和“概率”两个字。

事件不会发生的时候

如果事件发生的概率是 40%，不发生的几率就是 60%。这个自然的计算可以这样秒速：

$$P(\text{an event doesn't happen}) ~=~ 1 - P(\text{the event happens})$$

所有结果等可能的时候

如果你投掷一个普通的骰子，一个自然的假设是，所有六个面都是等可能的。 那么一个面出现的概率可以很容易地计算出来。 例如，骰子显示偶数的几率是：

$$\frac{\text{number of even faces}}{\text{number of all faces}} ~=~ \frac{\#\{2, 4, 6\}}{\#\{1, 2, 3, 4, 5, 6\}} ~=~ \frac{3}{6}$$

与之相似：

$$P(\text{die shows a multiple of 3}) ~=~ \frac{\#\{3, 6\}}{\#\{1, 2, 3, 4, 5, 6\}} ~=~ \frac{2}{6}$$

通常：

$$P(\text{an event happens}) ~=~ \frac{\#\{\text{outcomes that make the event happen}\}}{\#\{\text{all outcomes}\}}$$

前提是所有的结果都是等可能的。

并非所有的随机现象都像骰子一样简单。 下面的两个主要的概率规则甚至允许数学家在复杂的情况下找到概率。

两个事件必须同时发生时

假设你有一个盒子，包含三张纸条：一张红色，一张蓝色和一张绿色。 假设你随机抽两张纸条而不放回；也就是你把三张纸条打乱，抽一张，打乱其余两张，再从这两张中抽出一张。你先得到绿色纸条，然后是红色纸条的几率是多少？

有六种可能的颜色对：RB，BR，RG，GR，BG，GB（我们已经缩写了每种颜色的名字，就是它的第一个字母）。 所有这些都是抽样方案是等可能的，只有其中一个（GR）使事件发生。所以：

$$P(\text{green first, then red}) ~=~ \frac{\#\{\text{GR}\}}{\#\{\text{RB, BR, RG, GR, BG, GB}\}} ~=~ \frac{1}{6}$$

但是还有另外一种方法来得到答案，可以用两个阶段来思考这个事件。 必须首先抽取绿色纸条。几率是 1/3，也就是说在所有实验的大约 1/3 的重复中，先抽取了绿色纸条，但事件还没完成。在这 1/3 的重复中，必须再次抽取红色纸条。这个发生在大约 1/2 的重复中，所以：

$$P(\text{green first, then red}) ~=~ \frac{1}{2} \text{ of } \frac{1}{3} ~=~ \frac{1}{6}$$

这个计算通常按照事件顺序，像这样：

$$P(\text{green first, then red}) ~=~ \frac{1}{3} ~\times~ \frac{1}{2} ~=~ \frac{1}{6}$$

因数 1/2 叫做“假设第一次出现了绿色纸条，第二次出现红色纸条的条件几率”。

通常，我们拥有乘法规则：

$$P(\text{two events both happen}) ~=~ P(\text{one event happens}) \times P(\text{the other event happens, given that the first one happened})$$

两个事件同时发生的概率，等于第一个事件发生的概率，乘上第一个事件发生的情况下第二个事件发生的概率。

因此，这里有两个条件 - 一个事件必须发生，另一个也是 - 几率是分数的分数，这比两个因数的任何一个都要小。 满足的条件越多，满足的可能性就越小。

事件以两种不同的方式发生

相反，假设我们希望两张纸条中的一张是绿色的，另一张是红色的。 此事件不指定颜色必须出现的顺序。所以他们可以以任何顺序出现。

解决这样的问题的一个好方法就是对事件进行划分，以便它正好能够以几种不同的方式之一发生。“一绿一红”的自然划分是：GR，RG。

根据上面的计算，GR 和 RG 每个的几率都是 1/6。所以你可以通过把它们相加来计算一绿一红的概率。

$$P(\text{one green and one red}) ~=~ P(\text{GR}) + P(\text{RG}) ~=~ \frac{1}{6} + \frac{1}{6} ~=~ \frac{2}{6}$$

通常，我们拥有加法规则：

$$P(\text{an event happens}) ~=~ P(\text{first way it can happen}) + P(\text{second way it can happen})$$

事件发生的概率，等于以第一种方式发生的概率，加上以第二种方式发生的概率。

只要事件正好以两种方式之一发生。

因此，当事件以两种不同的方式之一发生时，发生的几率是一些几率的总和，因此比任何一种方式的几率都大。

乘法规则可以自然扩展到两个以上的事件，我们将在下面看到。 所以这个加法规则也有自然的扩展，事件可以以几种不同的方式之一发生。

我们将所有这些规则组合成示例，并用示例来结束该部分。

至少有一个成功

数据科学家经常使用来自总体的随机样本。 有时候问题就来了，就是总体中的一个特定个体选进样本的可能性。为了找出几率，这个个体被称为“成功”，问题是要找到样本包含成功的几率。

要看看如何计算这样的几率，我们从一个更简单的设定开始：投掷硬币两次。

如果你投掷硬币两次，有四个等可能的结果：HH，HT，TH 和 TT。 我们把正面缩写为 H ，反面缩写为 T。至少有一个正面的几率是 3/4。

得出这个答案的另一种方法是，弄清楚如果你不能得到至少一个正面，会发生什么事情：这两次投掷都必须是反面。所以：

$$P(\text{at least one head in two tosses}) ~=~ 1 - P(\text{both tails}) ~=~ 1 - \frac{1}{4} ~=~ \frac{3}{4}$$

要注意根据乘法规则：

$$P(\text{both tails}) ~=~ \frac{1}{4} ~=~ \frac{1}{2} \cdot \frac{1}{2} ~=~ \left(\frac{1}{2}\right)^2$$

这两个观察使我们能够在任何给定数量的投掷中找到至少一个正面的几率。 例如：

$$P(\text{at least one head in 17 tosses}) ~=~ 1 - P(\text{all 17 are tails}) ~=~ 1 - \left(\frac{1}{2}\right)^{17}$$

而现在我们有能力找到在骰子的投掷中，六点至少出现一次的几率：

$$P(\text{a single roll is not 6}) ~=~ P(1) + P(2) + P(3) + P(4) + P(5) ~=~ \frac{5}{6}$$

$$P(\text{at least one 6 in two rolls}) \;=\; 1 - P(\text{both rolls are not 6}) \;=\; 1 - \left(\frac{5}{6}\right)^2 and P(\text{at least one 6 in 17 rolls}) \;=\; 1 - \left(\frac{5}{6}\right)^{17}$$

下表展示了，这些概率随着投掷数量从 1 增加到 50 而增加。

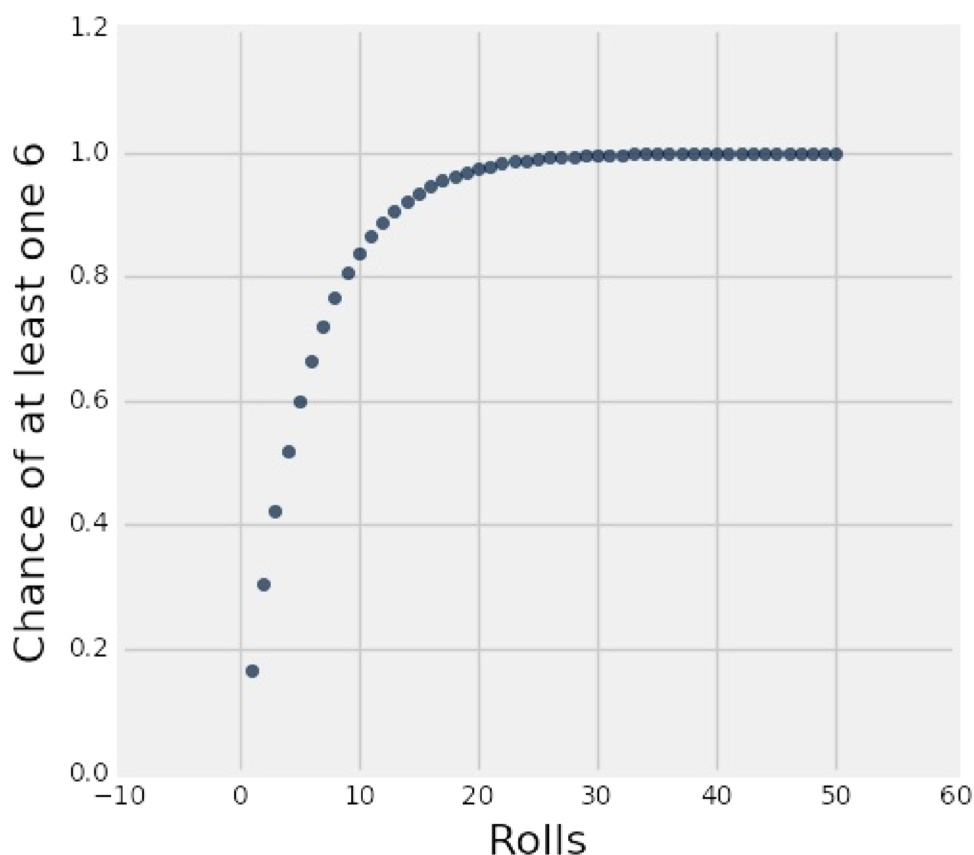
```
rolls = np.arange(1, 51, 1)
results = Table().with_columns(
    'Rolls', rolls,
    'Chance of at least one 6', 1 - (5/6)**rolls
)
results
```

Rolls	Chance of at least one 6
1	0.166667
2	0.305556
3	0.421296
4	0.517747
5	0.598122
6	0.665102
7	0.720918
8	0.767432
9	0.806193
10	0.838494

（省略了 40 行）

随着投掷数量的增加，六点至少出现一次的几率迅速增加。

```
results.scatter('Rolls')
```



在 50 次投掷中，你几乎肯定能得到至少一个六。

```
results.where('Rolls', are.equal_to(50))
```

Rolls	Chance of at least one 6
50	0.99989

像这样的计算可以用来找到，随机样本中选择特定个体的几率。 准确的计算将取决于抽样方案。 但是我们上面的观察的通常可以被推广：增加随机样本的大小增加了选择个体的几率。

抽样

现在我们来仔细看看抽样，例子基于 `top_movies.csv` 数据集。

```
top1 = Table.read_table('top_movies.csv')
top2 = top1.with_column('Row Index', np.arange(top1.num_rows))
top = top2.move_to_start('Row Index')

top.set_format(make_array(3, 4), NumberFormatter)
```

Row Index	Title	Studio	Gross	Gross (Adjusted)	Year
0	Star Wars: The Force Awakens	Buena Vista (Disney)	906,723,418	906,723,400	2015
1	Avatar	Fox	760,507,625	846,120,800	2009
2	Titanic	Paramount	658,672,302	1,178,627,900	1997
3	Jurassic World	Universal	652,270,625	687,728,000	2015
4	Marvel's The Avengers	Buena Vista (Disney)	623,357,910	668,866,600	2012
5	The Dark Knight	Warner Bros.	534,858,444	647,761,600	2008
6	Star Wars: Episode I - The Phantom Menace	Fox	474,544,677	785,715,000	1999
7	Star Wars	Fox	460,998,007	1,549,640,500	1977
8	Avengers: Age of Ultron	Buena Vista (Disney)	459,005,868	465,684,200	2015
9	The Dark Knight Rises	Warner Bros.	448,139,099	500,961,700	2012

（省略了 190 行）

对表格的行进行抽样

数据表的每一行代表一个个体；最重要的是，每个个体都是一部电影。 因此可以通过表格的行的抽样来实现对个体的抽样。

一行的内容是在同一个个体上测量的不同变量的值。 因此，行的内容的抽样形成了每个变量值的样本。

确定性样本

当你只是简单地指定，你要选择的集合中的哪些元素时，就不会涉及任何几率，可以创建确定性样本。

你已经做了很多次了，例如使用 `take` ：

```
top.take(make_array(3, 18, 100))
```

Row Index	Title	Studio	Gross	Gross (Adjusted)	Year
3	Jurassic World	Universal	652,270,625	687,728,000	2015
18	Spider-Man	Sony	403,706,375	604,517,300	2002
100	Gone with the Wind	MGM	198,676,459	1,757,788,200	1939

你也使用了`where`：

```
top.where('Title', are.containing('Harry Potter'))
```

Row Index	Title	Studio	Gross	Gross (Adjusted)	Year
22	Harry Potter and the Deathly Hallows Part 2	Warner Bros.	381,011,219	417,512,200	2011
43	Harry Potter and the Sorcerer's Stone	Warner Bros.	317,575,550	486,442,900	2001
54	Harry Potter and the Half-Blood Prince	Warner Bros.	301,959,197	352,098,800	2009
59	Harry Potter and the Order of the Phoenix	Warner Bros.	292,004,738	369,250,200	2007
62	Harry Potter and the Goblet of Fire	Warner Bros.	290,013,036	393,024,800	2005
69	Harry Potter and the Chamber of Secrets	Warner Bros.	261,988,482	390,768,100	2002
76	Harry Potter and the Prisoner of Azkaban	Warner Bros.	249,541,069	349,598,600	2004

虽然这些是电影的样本，它们并不涉及几率。

概率抽样

很多数据科学都根据随机样本中的数据得到结论。 根据随机样本的正确解释分析，需要数据科学家准确地检查随机样本。

总体是从中抽取样本的所有元素的集合。

概率样本是一种样本，在抽取样本之前，可以计算出的元素的任何子集将进入样本的几率。

在概率样本中，所有的元素不需要有相同的选中几率。

随机抽样方案

例如，假设根据以下方案，从三个个体 A，B 和 C 组成的总体中选择两个个体：

- 个体 A 选中概率为 1。
- 个体 B 或 C 根据掷硬币来选择：如果硬币为正面，选择 B，否则，选择 C。

这是一个大小为 2 的概率样本。下面是所有非空子集的选中几率：

```
A: 1
B: 1/2
C: 1/2
AB: 1/2
AC: 1/2
BC: 0
ABC: 0
```

个体 A 比 B 或 C 有更高的选中几率；的确，个体 A 肯定会被选中。由于这些差异是已知的和量化的，所以在处理样本时可以考虑这些差异。

系统样本

想象一下，总体的所有元素都列出在序列中。 抽样的一种方法是，先从列表中选择一个随机的位置，然后是它后面的等间隔的位置。样本由这些位置上的元素组成。这样的样本被称为系统样本。

在这里，我们将选择顶部一些行的系统样本。我们最开始随机选取前 10 行中的一行，然后我们将选取它后面的每个第 10 行。

```
"""Choose a random start among rows 0 through 9;
then take every 10th row."""

start = np.random.choice(np.arange(10))
top.take(np.arange(start, top.num_rows, 10))
```

Row Index	Title	Studio	Gross	Gross (Adjusted)	Year
6	Star Wars: Episode I - The Phantom Menace	Fox	474,544,677	785,715,000	1999
16	Iron Man 3	Buena Vista (Disney)	409,013,994	424,632,700	2013
26	Spider-Man 2	Sony	373,585,825	523,381,100	2004
36	Minions	Universal	336,045,770	354,213,900	2015
46	Iron Man 2	Paramount	312,433,331	341,908,200	2010
56	The Twilight Saga: New Moon	Sum.	296,623,634	338,517,700	2009
66	Meet the Fockers	Universal	279,261,160	384,305,300	2004
76	Harry Potter and the Prisoner of Azkaban	Warner Bros.	249,541,069	349,598,600	2004
86	The Exorcist	Warner Bros.	232,906,145	962,212,800	1973
96	Back to the Future	Universal	210,609,762	513,740,700	1985

（省略了 10 行）

运行单元个几次，看看输出如何变化。

这个系统样本是一个概率样本。 在这个方案中，所有的行都有机会被选中。 例如，当且仅当第 3 行被选中时，第 23 行才被选中，并且其几率是 1/10。

但并不是所有的子集都有相同的选中几率。 由于选中的行是等间隔的，大多数行的子集都没有机会被选中。 唯一可能的子集是由所有间隔为 10 的行构成的子集。任何这些子集都以 1/10 的几率被选中。 其他子集，如包含表格前 11 行的子集，选中几率都是 0。

放回或不放回的随机抽样

在这个课程中，我们将主要处理两个最直接的抽样方法。

首先是带放回的随机抽样，它（如我们前面所见）是 `np.random.choice` 从数组中抽样时的默认行为。

另一个称为“简单随机样本”，是随机抽取的样本，不带放回。在下一个个体被抽中之前，抽中的个体不会放回总体。例如，当你发牌时，就会发生这种抽样。

在下一章中，我们将使用模拟来研究带放回和不放回的大样本随机抽取。

绘制随机样本需要谨慎和精确。这不是随便的，即使这是“随机”一词的口语意义。如果你站在街头，选取前十名经过的人作为样本，你可能会认为你在随机抽样，因为你没有选择谁走过。但它不是一个随机样本 - 这是一个方便的例子。你没有提前知道每个人进入样本的概率，也许甚至你没有具体指定谁在总体中。

九、经验分布

> 原文：Empirical Distributions
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

大部分数据科学都涉及来自大型随机样本的数据。 在本节中，我们将研究这些样本的一些属性。

我们将从一个简单的实验开始：多次掷骰子并跟踪出现的点数。 `die` 表包含骰子面上的点数。 所有的数字只出现一次，因为我们假设骰子是平等的。

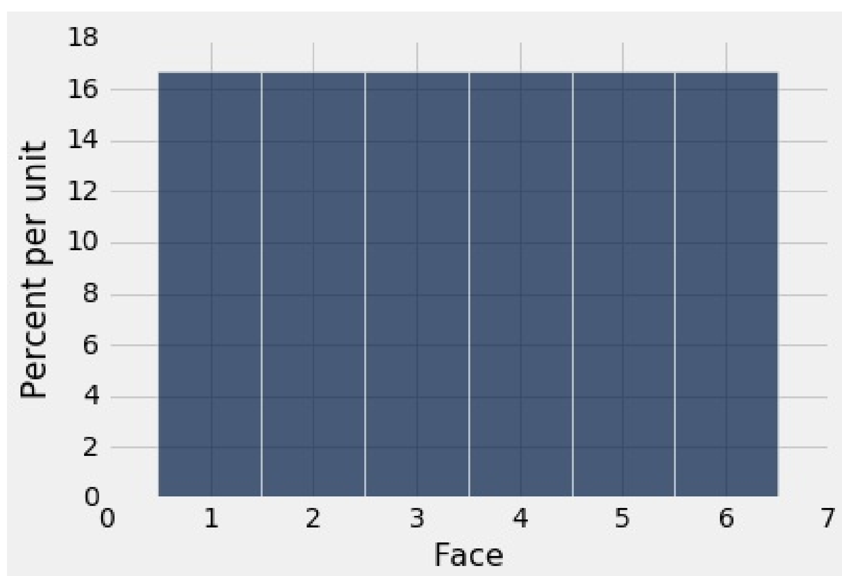
```
die = Table().with_column('Face', np.arange(1, 7, 1))
die
```

Face
1
2
3
4
5
6

概率分布

下面的直方图帮助我们可视化，每个面出现概率为 1/6 事实。 我们说直方图显示了所有可能的面的概率分布。 由于所有的条形表示相同的百分比几率，所以这个分布成为整数 1 到 6 上的均匀分布。

```
die_bins = np.arange(0.5, 6.6, 1)
die.hist(bins = die_bins)
```



递增值由相同的固定量分隔，例如骰子面上的值（递增值由 1 分隔），这样的变量被称为离散值。上面的直方图被称为离散直方图。它的桶由数组 `die_bins` 指定，并确保每个条形的中心是对应的整数值。

重要的是要记住，骰子不能显示 1.3 个点或 5.2 个点 - 总是显示整数个点。但是我们的可视化将每个值的概率扩展到条形区域。虽然在本课程的这个阶段这看起来有些随意，但是稍后当我们在离散直方图上叠加平滑曲线时，这将变得很重要。

在继续之前，让我们确保轴域上的数字是有意义的。每个面的概率是 1/6，四舍五入到小数点后两位的概率是 16.67%。每个桶的宽度是 1 个单位。所以每个条形的高度是每单位 16.67%。这与图形的水平和垂直比例一致。

经验分布

上面的分布由每个面的理论概率组成。 这不基于数据。 不投掷任何骰子，它就可以被研究和理解。

另一方面，经验分布是观测数据的分布。 他们可以通过经验直方图可视化。

让我们通过模拟一个骰子的投掷来获得一些数据。 这可以通过 1 到 6 的整数的带放回随机抽样来完成。为了使用 Python 来实现，我们将使用 `Table` 的 `sample` 方法，它带放回地随机抽取表中的行。它的参数是样本量，它返回一个由选定的行组成的表。`with_replacement=False` 的可选参数指定了应该抽取样本而不放回，但不适用于投掷骰子。

这是一个十次骰子投掷的结果。

```
die.sample(10)
```

Face
5
3
3
4
2
2
4
1
6
6

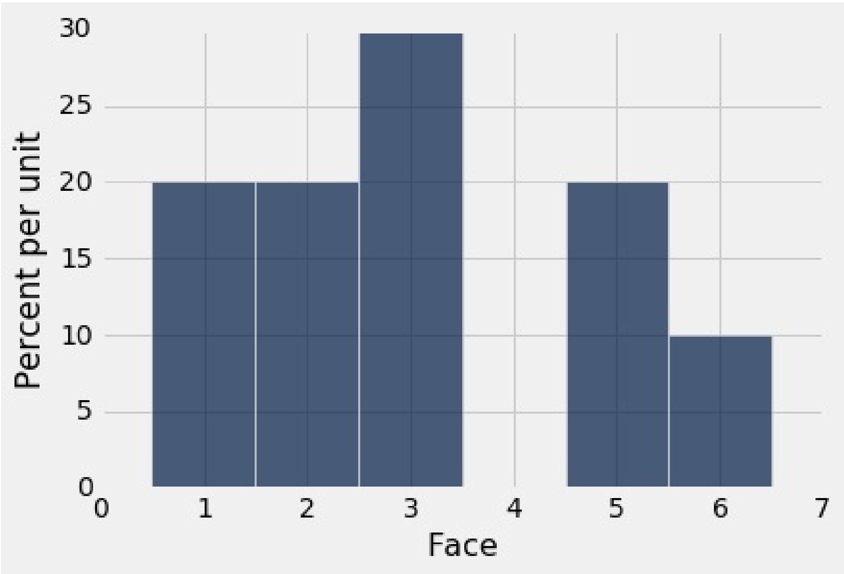
我们可以使用相同的方法来模拟尽可能多的投掷，然后绘制结果的经验直方图。 因为我们要反复这样做，所以我们定义了一个函数 `empirical_hist_die` ，它以样本量为参数；该函数根据其参数多次投掷骰子，然后绘制直方图。

```
def empirical_hist_die(n):
    die.sample(n).hist(bins = die_bins)
```

经验直方图

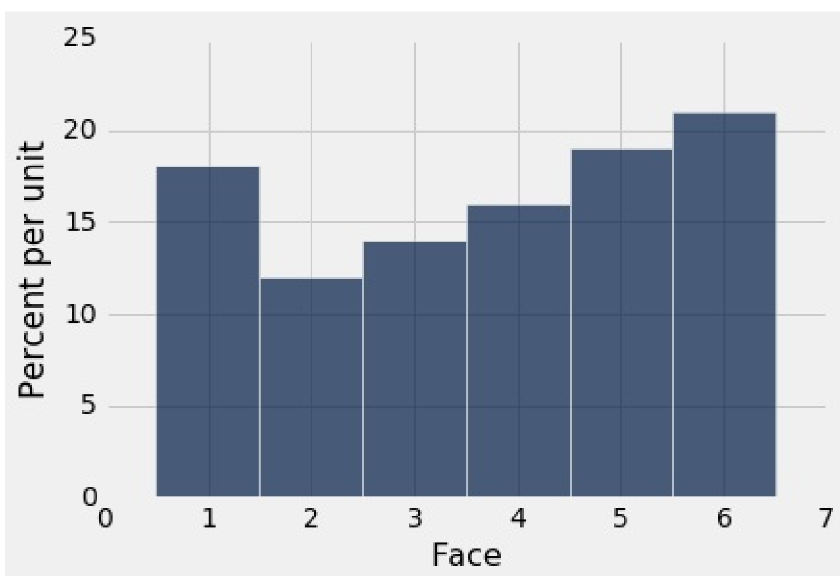
这是十次投掷的经验直方图。 它看起来不像上面的概率直方图。 运行该单元格几次，看看它如何变化。

```
empirical_hist_die(10)
```

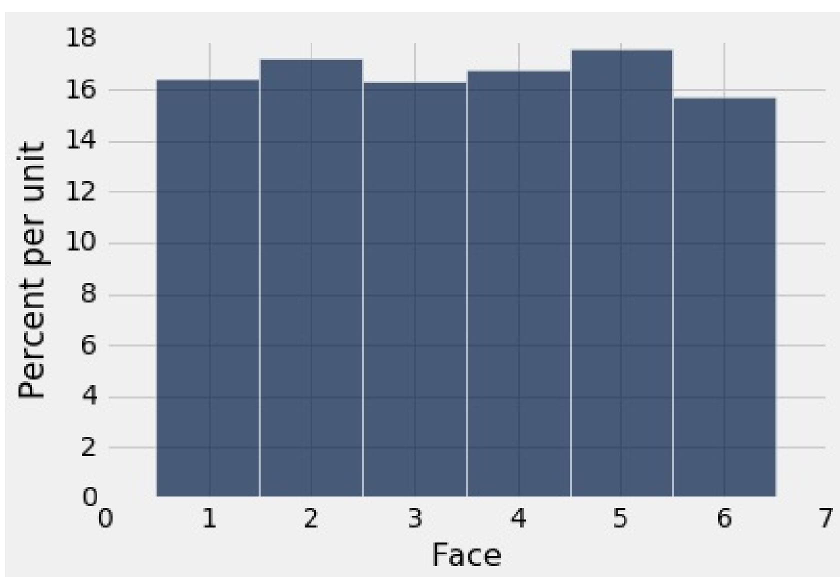


当样本量增加时，经验直方图开始看起来更像是理论概率的直方图。

```
empirical_hist_die(100)
```




```
empirical_hist_die(1000)
```



当我们增加模拟中的投掷次数时，每个条形的面积接近 16.67%，这是概率直方图中每个条形的面积。

我们在实例中观察到了一般规则：

平均定律

如果偶然的实验在相同的条件下独立重复，那么从长远来看，事件发生的频率越来越接近事件的理论概率。

例如，从长远来看，四点的比例越来越接近 1/6。

这里“独立地且在相同的条件下”意味着，无论所有其他重复的结果如何，每个重复都以相同的方式执行。

从总体中取样

当随机样本来自较大总体时，平均定律也成立。

作为一个例子，我们将研究航班延误时间的总体。 `united` 表包含 2015 年夏天从旧金山出发的美联航国内航班的数据。数据由美国运输部运输统计局公布。

这里有 13,825 行，每行对应一个航班。 列是航班日期，航班号，目的地机场代码和以分钟为单位的出发延误时间。有些延误时间是负的；那些航班提前离开。

```
united = Table.read_table('united_summer2015.csv')
united
```

Date	Flight Number	Destination	Delay
6/1/15	73	HNL	257
6/1/15	217	EWR	28
6/1/15	237	STL	-3
6/1/15	250	SAN	0
6/1/15	267	PHL	64
6/1/15	273	SEA	-6
6/1/15	278	SEA	-8
6/1/15	292	EWR	12
6/1/15	300	HNL	20
6/1/15	317	IND	-10

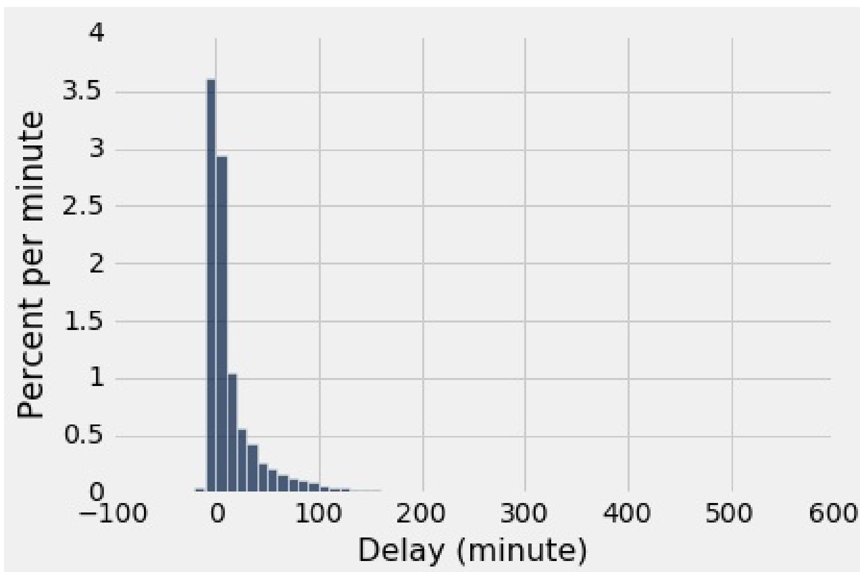
（省略了 13815 行）

一个航班提前 16 分钟起飞，另一个航班延误 580 分钟。 其他延迟时间几乎都在 -10 分钟到 200 分钟之间，如下面的直方图所示。

```
united.column('Delay').min()
-16

united.column('Delay').max()
580

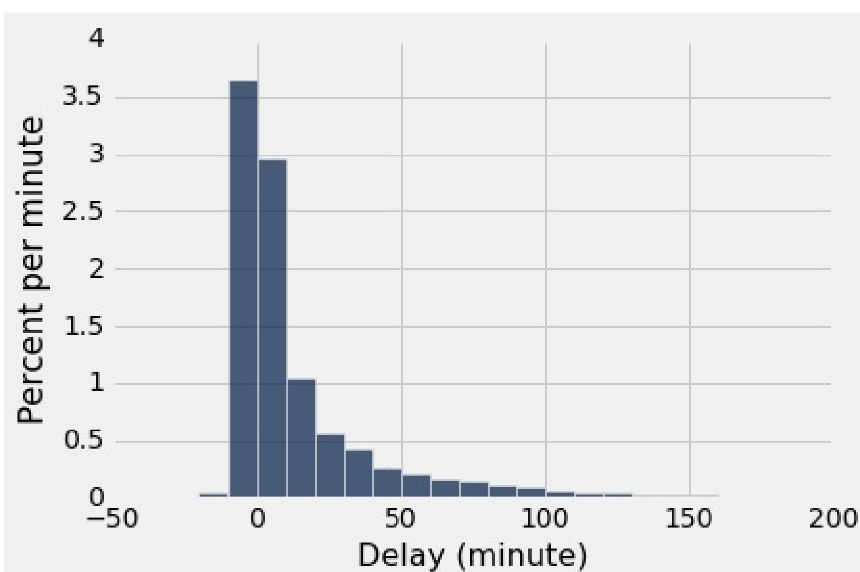
delay_bins = np.append(np.arange(-20, 301, 10), 600)
united.select('Delay').hist(bins = delay_bins, unit = 'minute')
```



就本节而言，仅仅关注部分数据就足够了，我们忽略延迟超过 200 分钟的 0.8% 的航班。 这个限制只是为了视觉便利。 该表仍然保留所有的数据。

```
united.where('Delay', are.above(200)).num_rows/united.num_rows
0.008390596745027125

delay_bins = np.arange(-20, 201, 10)
united.select('Delay').hist(bins = delay_bins, unit = 'minute')
```



`[0,10)` 的条形高度不到每分钟 3%，这意味着只有不到 30% 的航班延误了 0 到 10 分钟。 这是通过行的计数来确认的：

```
united.where('Delay', are.between(0, 10)).num_rows/united.num_rows
0.2935985533453888
```

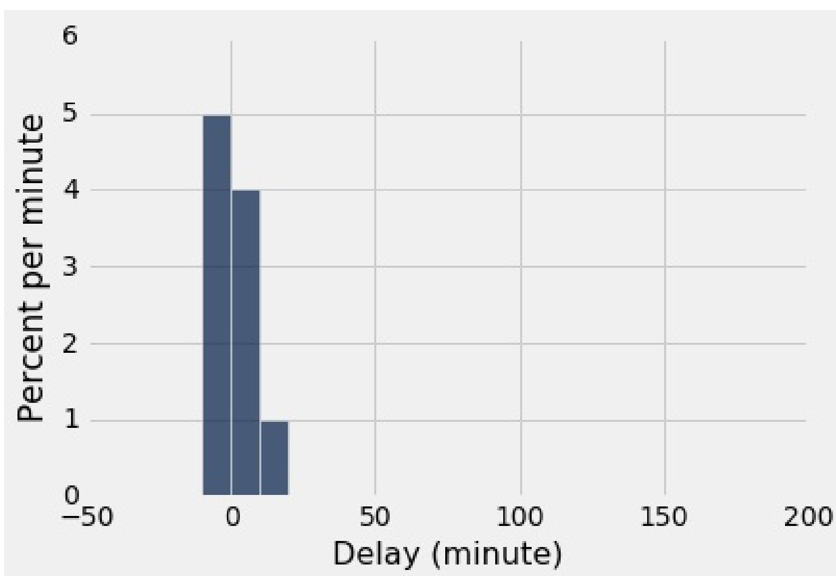
样本的经验分布

现在让我们将这 13,825 个航班看做一个总体，并从中带放回地抽取随机样本。 将我们的分析代码打包成一个函数是有帮助的。 函数 `empirical_hist_delay` 以样本量为参数，绘制结果的经验直方图。

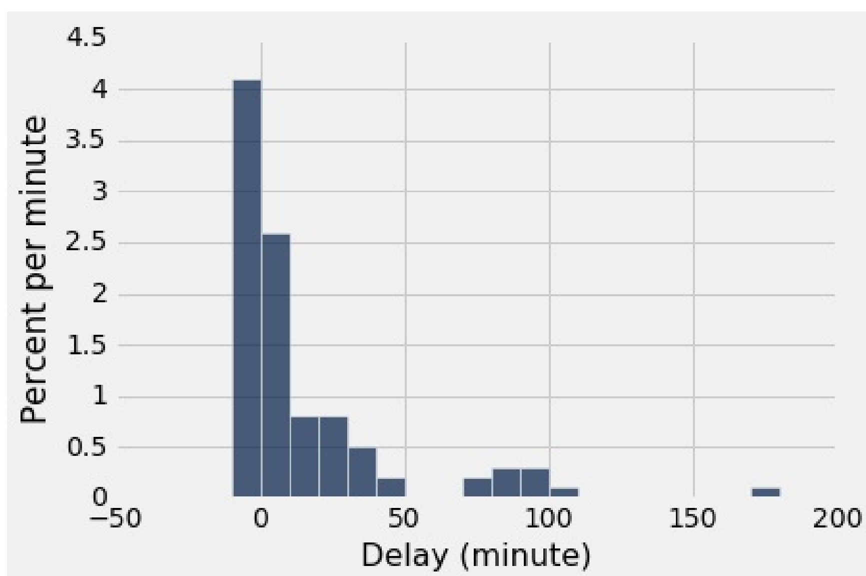
```
def empirical_hist_delay(n):
    united.sample(n).select('Delay').hist(bins = delay_bins, unit = 'minute')
```

正如我们用骰子所看到的，随着样本量的增加，样本的经验直方图更接近于总体的直方图。 将这些直方图与上面的总体直方图进行比较。

```
empirical_hist_delay(10)
```

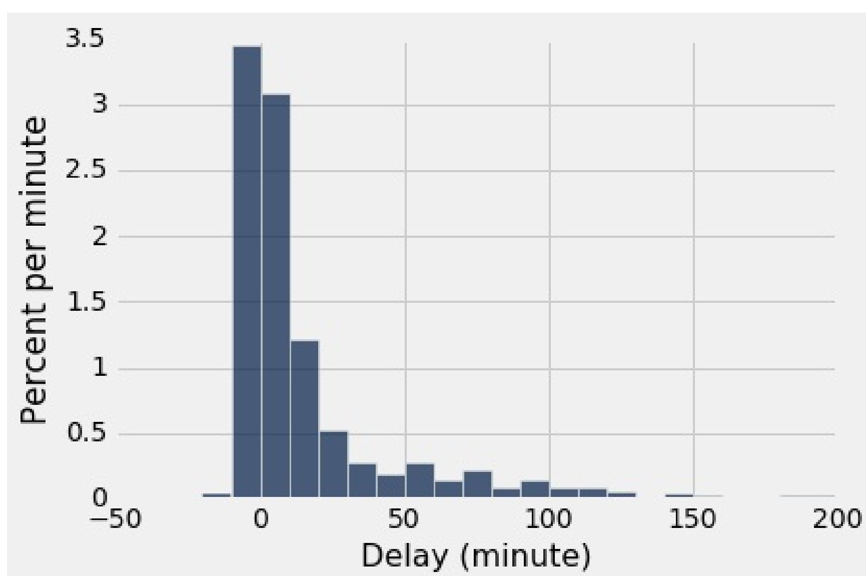



```
empirical_hist_delay(100)
```



最一致的可见差异在总体中罕见的值之中。 在我们的示例中，这些值位于分布的最右边。 但随着样本量的增加，这些值以大致正确的比例，开始出现在样本中。

```
empirical_hist_delay(1000)
```



样本的经验直方图的总结

我们在本节中观察到的东西，可以总结如下：

对于大型随机样本，样本的经验直方图类似于总体的直方图，概率很高。

这证明了，在统计推断中使用大型随机样本是合理的。 这个想法是，由于大型随机样本可能类似于从中抽取的总体，从样本中计算出的数量可能接近于总体中相应的数量。

轮盘赌

上面的分布让我们对整个随机样本有了印象。但有时候我们只是对基于样本计算的一个或两个量感兴趣。

例如，假设样本包含一系列投注的输赢。那么我们可能只是对赢得的总金额感兴趣，而不是输赢的整个序列。

使用我们的几率长期行为的新知识，让我们探索赌博游戏。我们将模拟轮盘赌，它在拉斯维加斯和蒙特卡洛等赌场中受欢迎。

在内华达，轮盘赌的主要随机器是一个带有 38 个口袋的轮子。其中两个口袋是绿色的，十八个黑色，十八个红色。轮子在主轴上，轮子上有一个小球。当轮子旋转时，球体跳起来，最后落在其中一个口袋里。这就是获胜的口袋。

`wheel`表代表内华达轮盘赌的口袋。

```
wheel
```

Pocket	Color
0	green
00	green
1	red
2	black
3	red
4	black
5	red
6	black
7	red
8	black

（省略了 28 行）

你可以对轮盘赌桌上展示的几个预先指定的口袋下注。 如果你对“红色”下注，如果球落在红色的口袋里，你就赢了。

红色的下注返回相等的钱。 也就是说，它支付一比一。为了理解这是什么意思，假设你在“红色”下注一美元。 第一件事情发生之前，即使在车轮旋转之前，你必须交出你的一美元。 如果球落在绿色或黑色的口袋里，你就失去它了。 如果球落在红色的口袋里，你会把你的钱拿回来（让你不输不赢），再加上另外一美元的奖金。

函数 `red_winnings` 以一个颜色作为参数，如果颜色是红色，则返回 `1` 。 对于所有其他颜色，它返回 `-1` 。 我们将 `red_winnings` 应用于 `wheel` 的 `Color` 列，来获得新的表 `bets` ，如果你对红色下注一美元，它显示每个口袋的净收益。

```
def red_winnings(color):
    if color == 'red':
        return 1
    else:
        return -1
bets = wheel.with_column(
    'Winnings: Red', wheel.apply(red_winnings, 'Color')
    )
bets
```

Pocket	Color	Winnings: Red
0	green	-1
00	green	-1
1	red	1
2	black	-1
3	red	1
4	black	-1
5	red	1
6	black	-1
7	red	1
8	black	-1

（省略了 28 行）

假设我们决定对红色下注一美元，会发生什么呢？

这里是一轮的模拟。

```
one_spin = bets.sample(1)
one_spin
```

Pocket	Color	Winnings: Red
14	red	1

这轮的颜色是 `Color` 列中的值。 无论你的赌注如何，结果可能是红色，绿色或黑色。 要看看这些事件发生的频率，我们可以模拟许多这样的单独轮次，并绘制出我们所看到的颜色的条形图。 （我们可以称之为经验条形图。）

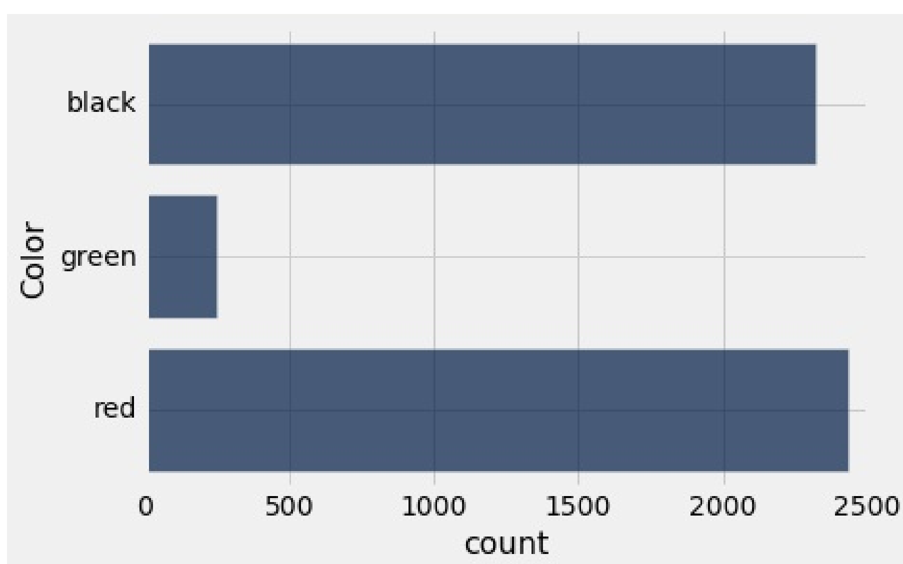
为了实现它，我们可以使用 `for` 循环。 我们在这里选择了重复 5000 次，但是当你运行这个单元格时，你可以改变它。

```
num_simulations = 5000

colors = make_array()
winnings_on_red = make_array()

for i in np.arange(num_simulations):
    spin = bets.sample(1)
    new_color = spin.column("Color").item(0)
    colors = np.append(colors, new_color)
    new_winnings = spin.column('Winnings: Red')
    winnings_on_red = np.append(winnings_on_red, new_winnings)

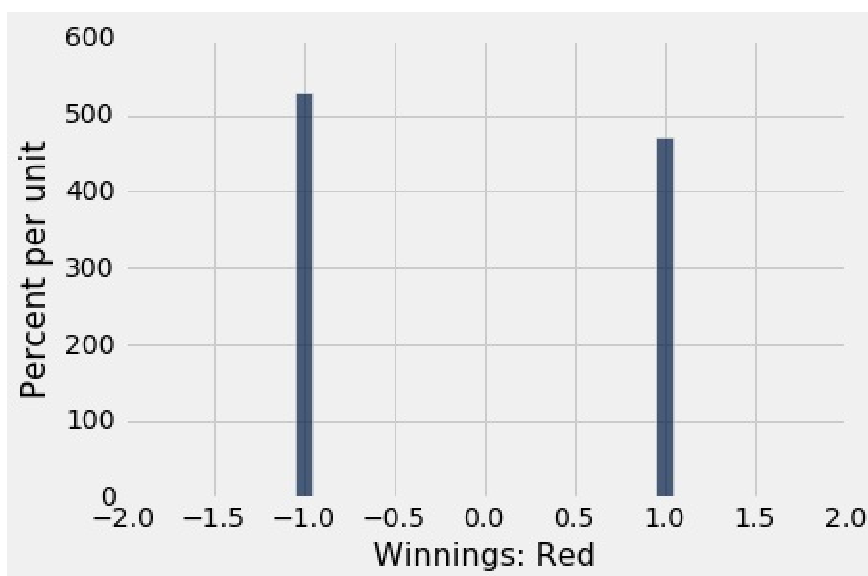
Table().with_column('Color', colors)\
       .group('Color')\
       .barh('Color')
```



38 个口袋里有 18 个是红色的，每个口袋都是等可能的。 因此，在 5000 次模拟中，我们预计大致（但可能不是完全）看到 `18/38*5000` 或者 2,368 次红色。模拟证明了这一点。

在模拟中，我们也记录了你的奖金。 这些经验直方图显示了，你对红色下注的不同结果的（近似）几率。

```
Table().with_column('Winnings: Red', winnings_on_red)\
       .hist(bins = np.arange(-1.55, 1.65, .1))
```



每个模拟的唯一可能的结果是，你赢了一美元或输了一美元，这反映在直方图中。 我们也可以看到，你赢的次数要比输的次数少一点。 你喜欢这个赌博策略吗？

多次游戏

大多数轮盘赌玩家玩好几轮。 假设你在 200 次独立轮次反复下注一美元。 你总共会赚多少钱？

这里是一套 200 轮的模拟。 `spins` 表包括所有 200 个赌注的结果。 你的净收益是 `Winnings: Red` 列中所有 +1 和 -1 的和。

```
spins = bets.sample(200)
spins.column('Winnings: Red').sum()
-26
```

运行几次单元格。 有时你的净收益是正的，但更多的时候它似乎是负的。

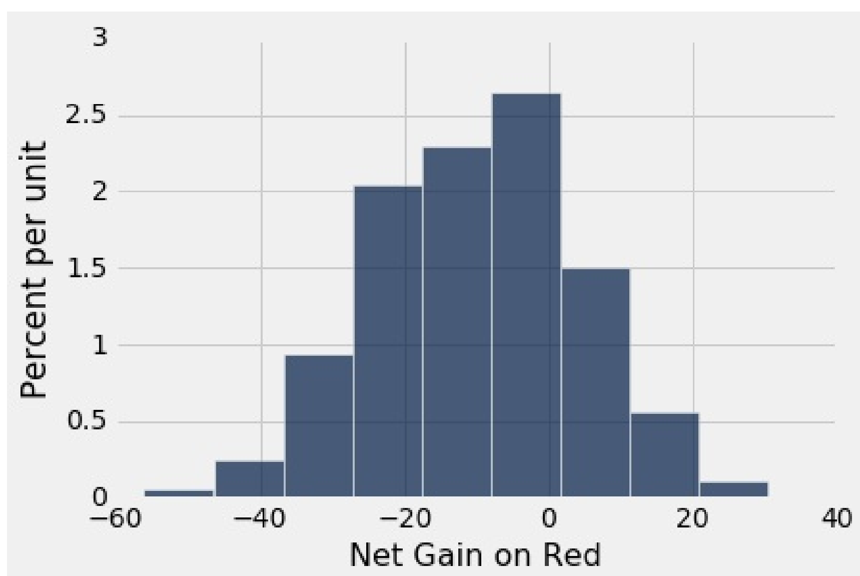
为了更清楚地看到发生了什么，让我们多次模拟 200 轮，就像我们模拟一轮那样。 对于每次模拟，我们将记录来自 200 轮的总奖金。 然后我们将制作 5000 个不同的模拟总奖金的直方图。

```
num_spins = 200

net_gain = make_array()

for i in np.arange(num_simulations):
    spins = bets.sample(num_spins)
    new_net_gain = spins.column('Winnings: Red').sum()
    net_gain = np.append(net_gain, new_net_gain)

Table().with_column('Net Gain on Red', net_gain).hist()
```



注意横轴上 0 的位置。 这就是你不赚不赔的地方。 通过使用这个赌博策略，你喜欢这个赚钱几率吗？

如果对红色下注不吸引人，也许值得尝试不同的赌注。“分割”（Split）是轮盘赌桌上两个相邻号码的下注，例如 0 和 00。分割的回报是 17 比 1。

`split_winnings` 函数将口袋作为参数，如果口袋是 0 或 00，则返回 17。对于所有其他口袋，返回 -1。

表格 `more_bets` 是投注表格的一个版本，扩展的一列是对 0/00 分割下注的情况下，每个口袋的奖金。

```
def split_winnings(pocket):
    if pocket == '0':
        return 17
    elif pocket == '00':
        return 17
    else:
        return -1
more_bets = wheel.with_columns(
    'Winnings: Red', wheel.apply(red_winnings, 'Color'),
    'Winnings: Split', wheel.apply(split_winnings, 'Pocket')
    )
more_bets
```

Pocket	Color	Winnings: Red	Winnings: Split
0	green	-1	17
00	green	-1	17
1	red	1	-1
2	black	-1	-1
3	red	1	-1
4	black	-1	-1
5	red	1	-1
6	black	-1	-1
7	red	1	-1
8	black	-1	-1

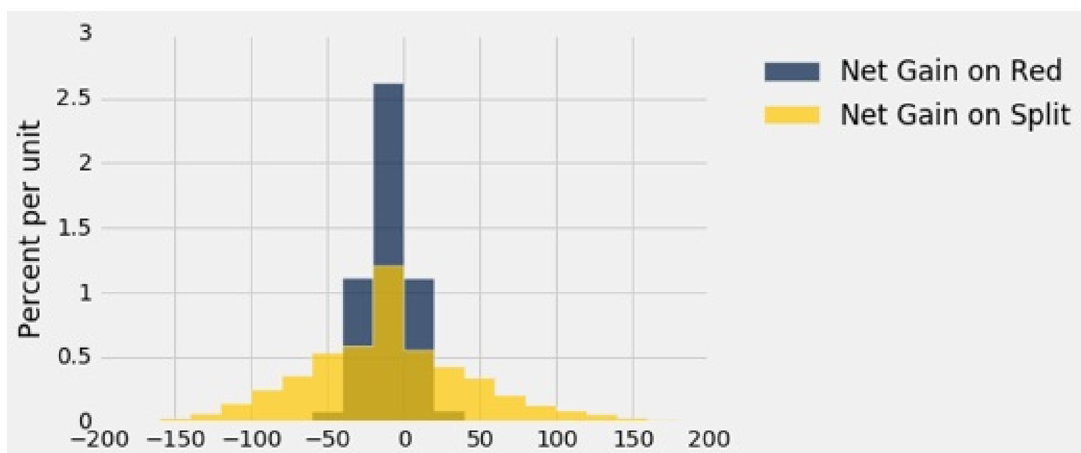
（省略了 28 行）

下面的代码模拟了两个投注的结果 - 红色和 0/00 分割 - 在 200 轮中。 代码与以前的模拟相同，除了添加了 Split。（注意：`num_simulations` 和 `num_spins` 之前分别定义为 5,000 和 200，所以我们不需要再次定义它们。）

```
net_gain_red = make_array()
net_gain_split = make_array()

for i in np.arange(num_simulations):
    spins = more_bets.sample(num_spins)
    new_net_gain_red = spins.column('Winnings: Red').sum()
    net_gain_red = np.append(net_gain_red, new_net_gain_red)
    new_net_gain_split = spins.column('Winnings: Split').sum()
    net_gain_split = np.append(net_gain_split, new_net_gain_split)

Table().with_columns(
    'Net Gain on Red', net_gain_red,
    'Net Gain on Split', net_gain_split
    ).hist(bins=np.arange(-200, 200, 20))
```



横轴上 0 的位置表明，无论你选择哪种赌注，你都更有可能赔钱而不是赚钱。在两个直方图中，不到 50% 的区域在 0 的右侧。

然而，分割的赌注赚钱几率更大，赚取超过 50 美元的机会也是如此。 金色直方图有很多区域在五十美元的右侧，而蓝色直方图几乎没有。 那么你应该对分割下注吗？

这取决于你愿意承担多少风险，因为直方图还表明，如果你对分割下注，你比对红色下注更容易损失超过 50 美元。

轮盘赌桌上，所有赌注的单位美元的预期净损失相同（除了线注，这是更糟的）。 但一些赌注的回报比其他赌注更为可变。 你可以选择这些赌注，只要你准备好可能会大输一场。

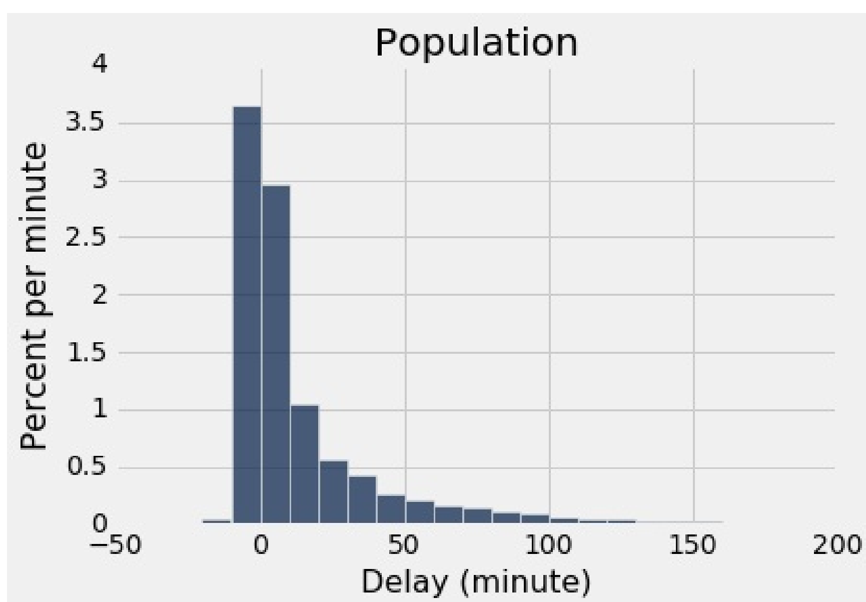
统计量的经验分布

平均定律意味着，大型随机样本的经验分布类似于总体的分布，概率相当高。

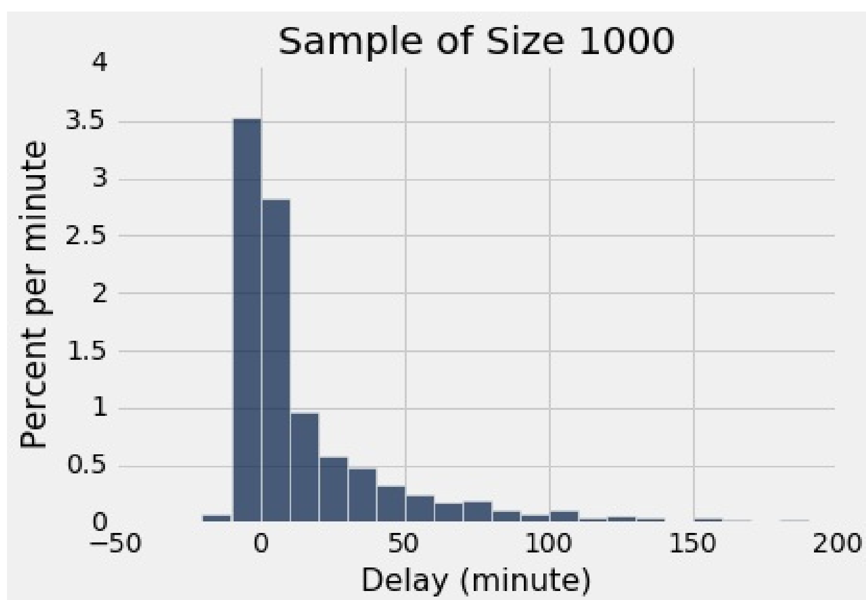
在两个直方图中可以看到相似之处：大型随机样本的经验直方图很可能类似于总体的直方图。

提醒一下，这里是所有美联航航班延误的直方图，以及这些航班的大小为 1000 的随机样本的经验直方图。

```
united = Table.read_table('united_summer2015.csv')
delay_bins = np.arange(-20, 201, 10)
united.select('Delay').hist(bins = delay_bins, unit = 'minute')
plots.title('Population');
```




```
sample_1000 = united.sample(1000)
sample_1000.select('Delay').hist(bins = delay_bins, unit = 'minute')
plots.title('Sample of Size 1000');
```



两个直方图明显相似，虽然他们并不等价。

参数

我们经常对总体相关的数量感兴趣。

在选民的总体中，有多少人会投票给候选人 A 呢？ 在 Facebook 用户的总体中，用户最多拥有的 Facebook 好友数是多少？ 在美联航航班的总体中，起飞延误时间的中位数是多少？

与总体相关的数量被称为参数。 对于美联航航班的总体，我们知道参数“延误时间的中位数”的值：

```
np.median(united.column('Delay'))
2.0
```

NumPy 函数 `median` 返回数组的中值（中位数）。 在所有的航班中，延误时间的中位数为 2 分钟。 也就是说，总体中约有 50% 的航班延误了 2 分钟以内：

```
united.where('Delay', are.below_or_equal_to(2)).num_rows/united.num_rows
0.5018444846292948
```

一半的航班在预定起飞时间的 2 分钟之内起飞。 这是非常短暂的延误！

注意。 由于“重复”，百分比并不完全是 50，也就是说，延误了 2 分钟的航班有 480 个。数据集中的重复很常见，我们不会在这个课程中担心它。

```
united.where('Delay', are.equal_to(2)).num_rows
480
```

统计

在很多情况下，我们会感兴趣的是找出未知参数的值。 为此，我们将依赖来自总体的大型随机样本的数据。

统计量（注意是单数！）是使用样本中数据计算的任何数字。 因此，样本中位数是一个统计量。

请记住，`sample_1000` 包含来自 `united` 的 1000 个航班的随机样本。 样本中位数的观测值是：

```
np.median(sample_1000.column('Delay'))
2.0
```

我们的样本 - 一千个航班 - 给了我们统计量的观测值。 这提出了一个重要的推论问题：

统计量的数值可能会有所不同。 使用基于随机样本的任何统计量时，首先考虑的事情是，样本可能不同，因此统计量也可能不同。

```
np.median(united.sample(1000).column('Delay'))
3.0
```

运行单元格几次来查看答案的变化。 通常它等于 2，与总体参数值相同。 但有时候不一样。

统计量有多么不同？ 回答这个问题的一种方法是多次运行单元格，并记下这些值。 这些值的直方图将告诉我们统计量的分布。

我们将使用 `for` 循环来“多次运行单元格”。 在此之前，让我们注意模拟中的主要步骤。

模拟统计量

我们将使用以下步骤来模拟样本中位数。 你可以用任何其他样本量来替换 1000 的样本量，并将样本中位数替换为其他统计量。

第一步：生成一个统计量。 抽取大小为 1000 的随机样本，并计算样本的中位数。 注意中位数的值。

第二步：生成更多的统计值。 重复步骤 1 多次，每次重新抽样。

第三步：结果可视化。 在第二步结束时，你将会记录许多样本中位数，每个中位数来自不同的样本。 你可以在表格中显示所有的中位数。 你也可以使用直方图来显示它们 - 这是统计量的经验直方图。

我们现在执行这个计划。 正如在所有的模拟中，我们首先创建一个空数组，我们在其中收集我们的结果。

- 上面的第一步是 `for` 循环的主体。
- 第二步，重复第一步“无数次”，由循环完成。 我们“无数次”是5000次，但是你可以改变这

个。
- 第三步是显示表格，并在后面的单元格中调用 `hist`。

该单元格需要大量的时间来运行。 那是因为它正在执行抽取大小为 1000 的样本，并计算其中位数的过程，重复 5000 次。 这是很多抽样和重复！

```
medians = make_array()

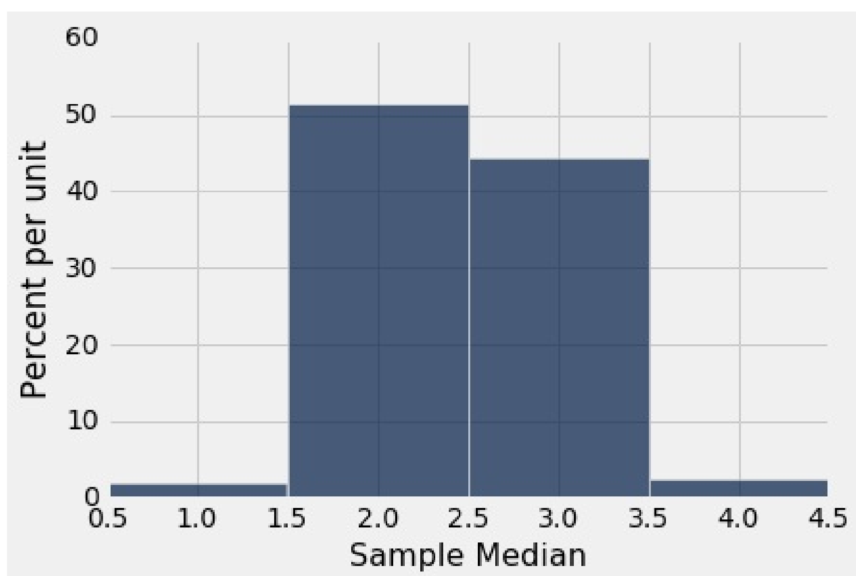
for i in np.arange(5000):
    new_median = np.median(united.sample(1000).column('Delay'))
    medians = np.append(medians, new_median)

Table().with_column('Sample Median', medians)
```

Sample Median
3
2
2
3
2
2
2
3
1
3

（省略了 4990 行）

```
Table().with_column('Sample Median', medians).hist(bins=np.arange(0.5, 5, 1))
```



你可以看到样本中位数很可能接近 2，这是总体中位数的值。 由于 1000 次航班延误的样本可能与延误总体相似，因此这些样本的延误中位数应接近总体的延误中位数，也就不足为奇了。

这是一个例子，统计量如何较好估计参数。

模拟的威力

如果我们能够生成所有可能的大小为 1000 的随机样本，我们就可以知道所有可能的统计量（样本中位数），以及所有这些值的概率。我们可以在统计量的概率直方图中可视化所有值和概率。

但在许多情况下（包括这个），所有可能的样本数量足以超过计算机的容量，概率的纯粹数学计算可能有些困难。

这是经验直方图的作用。

我们知道，如果样本量很大，并且如果重复抽样过程无数次，那么根据平均定律，统计量的经验直方图可能类似于统计量的概率直方图。

这意味着反复模拟随机过程是一种近似概率分布的方法，不需要在数学上计算概率，或者生成所有可能的随机样本。因此，计算机模拟成为数据科学中的一个强大工具。他们可以帮助数据科学家理解随机数量的特性，这些数据会以其他方式进行分析。

这就是这种的模拟的经典例子。

估计敌军飞机的数量

在第二次世界大战中，为盟军工作的数据分析师负责估算德国战机的数量。 这些数据包括盟军观察到的德国飞机的序列号。 这些序列号为数据分析师提供了答案。

为了估算战机总数，数据分析人员必须对序列号做出一些假设。 这里有两个这样的假设，大大简化，使我们的计算更容易。

- 战机有 `N` 架，编号为 `1,2, ..., N` 。
- 观察到的飞机从 `N` 架飞机中均匀、随机带放回地抽取。

目标是估计数字 `N` 。 这是未知的参数。

假设你观察一些飞机并记下他们的序列号。 你如何使用这些数据来猜测 `N` 的值？ 用于估计的自然和简单的统计量，就是观察到的最大的序列号。

让我们看看这个统计量如何用于估计。 但首先是另一个简化：现在一些历史学家估计，德国的飞机制造业生产了近 10 万架不同类型的战机，但在这里我们只能想象一种。 这使得假设 1 更易于证明。

假设实际上有 `N = 300` 个这样的飞机，而且你观察到其中的 30 架。 我们可以构造一个名为 `serialno` 的表，其中包含序列号 `1` 到 `N` 。 然后，我们可以带放回取样 30 次（见假设 2），来获得我们的序列号样本。 我们的统计量是这 30 个数字中的最大值。 这就是我们用来估计参数 `N` 的东西。

```
N = 300
serialno = Table().with_column('serial Number', np.arange(1, N+1))
serialno
```

serial number
1
2
3
4
5
6
7
8
9
10

（省略了 290 行）

```
serialno.sample(30).column(0).max()
291
```

与所有涉及随机抽样的代码一样，运行该单元几次；来查看变化。你会发现，即使只有 300 个观测值，最大的序列号通常在 250-300 范围内。

原则上，最大的序列号可以像 1 那样小，如果你不幸看到了 30 次 1 号机。如果你至少观察到一次 300 号机，它可能会增大到 300。但通常情况下，它似乎处于非常高的 200 以上。看起来，如果你使用最大的观测序列号作为你对总数的估计，你不会有太大的错误。

模拟统计

让我们模拟统计，看看我们能否证实它。模拟的步骤是：

第一步。从 1 到 300 带放回地随机抽样 30 次，并注意观察到的最大数量。这是统计量。

第二步。重复步骤一 750 次，每次重新取样。你可以用任何其他的大数值代替 750。

第三步。创建一个表格来显示统计量的 750 个观察值，并使用这些值绘制统计量的经验直方图。

```
sample_size = 30
repetitions = 750
maxes = make_array()

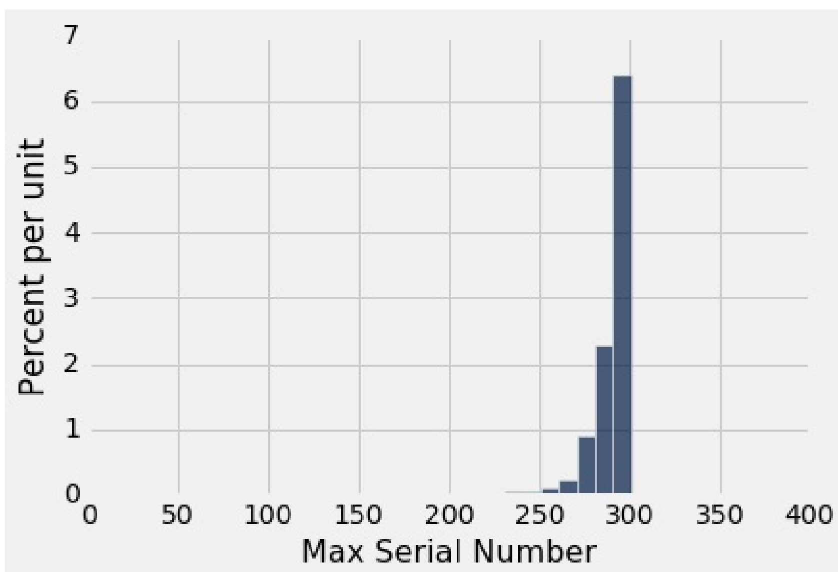
for i in np.arange(repetitions):
    sampled_numbers = serialno.sample(sample_size)
    maxes = np.append(maxes, sampled_numbers.column(0).max())

Table().with_column('Max Serial Number', maxes)
```

Max Serial Number
280
253
294
299
298
237
296
297
293
295

（省略了 740 行）

```
every_ten = np.arange(1, N+100, 10)
Table().with_column('Max Serial Number', maxes).hist(bins = every_ten)
```



这是 750 个估计值的直方图，每个估计值是统计量“观察到的最大序列号”的观测值。

正如你所看到的，尽管在理论上它们可能会小得多，但估计都在 300 附近。直方图表明，作为飞机总数的估计，最大的序列号可能低了大约 10 到 25 个。但是，飞机的真实数量低了 50 个是不太可能的。

良好的近似

我们前面提到过，如果生成所有可能的样本，并计算每个样本的统计量，那么你将准确了解统计量可能有多么不同。事实上，你将会完整地列举统计量的所有可能值及其所有概率。

换句话说，你将得到统计量的概率分布和概率直方图。

统计量的概率分布也称为统计量的抽样分布，因为它基于所有可能的样本。

但是，我们上面已经提到，可能的样本总数往往非常大。例如，如果有 300 架飞机，你可以看到的，30 个序列号的可能序列总数为：

```
300**30
205891132094649000000000000000000000000000000000000000000000000000000000000
```

这是很多样本。 幸运的是，我们不必生成所有这些。 我们知道统计量的经验直方图，基于许多但不是全部可能的样本，是概率直方图的很好的近似。 因此统计量的经验分布让我们很好地了解到，统计量可能有多么不同。

确实，统计量的概率分布包含比经验分布更准确的统计量信息。 但是，正如在这个例子中一样，通常经验分布所提供的近似值，足以让数据科学家了解统计量可以变化多少。 如果你有一台计算机，经验分布更容易计算。 因此，当数据科学家试图理解统计的性质时，通常使用

经验分布而不是精确的概率分布。

参数的不同估计

这里举一个例子来说明这一点。 到目前为止，我们已经使用了最大的观测序号作为飞机总数的估计。 但还有其他可能的估计，我们现在将考虑其中之一。

这个估计的基本思想是观察到的序列号的平均值可能在1到`N`之间。 因此，如果`A`是平均值，那么：

$$A \approx \frac{N}{2} \quad \text{and so} \quad N \approx 2A$$

因此，可以使用一个新的统计量化来估计飞机总数：取观测到的平均序列号并加倍。

与使用最大的观测数据相比，这种估计方法如何？ 计算新统计量的概率分布并不容易。 但是和以前一样，我们可以模拟它来近似得到概率。 我们来看看基于重复抽样的统计量的经验分布。 为了便于比较，重复次数选择为 750，与之前的模拟相同。

```
maxes = make_array()
twice_ave = make_array()

for i in np.arange(repetitions):
    sampled_numbers = serialno.sample(sample_size)

    new_max = sampled_numbers.column(0).max()
    maxes = np.append(maxes, new_max)

    new_twice_ave = 2*np.mean(sampled_numbers.column(0))
    twice_ave = np.append(twice_ave, new_twice_ave)


results = Table().with_columns(
    'Repetition', np.arange(1, repetitions+1),
    'Max', maxes,
    '2*Average', twice_ave
)

results
```

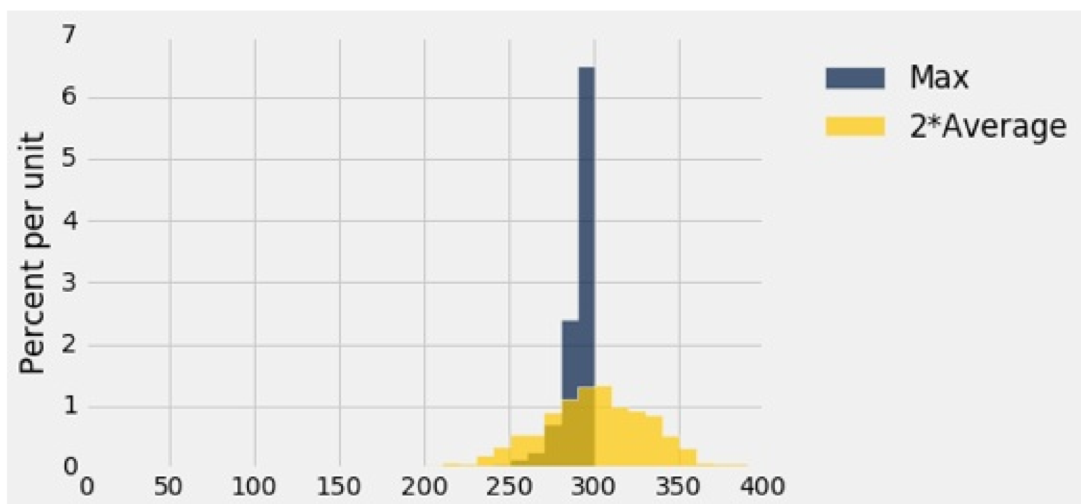
Repetition	Max	2*Average
1	296	312.067
2	283	290.133
3	290	250.667
4	296	306.8
5	298	335.533
6	281	240
7	300	317.267
8	295	322.067
9	296	317.6
10	299	308.733

（省略了 740 行）

请注意，与所观察到的最大数字不同，新的估计值（“平均值的两倍”）可能会高估飞机的数量。 当观察到的序列号的平均值接近于 `N` 而不是 `1` 时，就会发生这种情况。

下面的直方图显示了两个估计的经验分布。

```
results.drop(0).hist(bins = every_ten)
```



你可以看到，原有方法几乎总是低估; 形式上，我们说它是有偏差的。 但它的变异性很小，很可能接近真正的飞机总数。

新方法高估了它，和低估的频率一样，因此从长远来看，平均而言大致没有偏差。 然而，它比旧的估计更可变，因此容易出现较大的绝对误差。

这是一个偏差 - 变异性权衡的例子，在竞争性估计中并不罕见。 你决定使用哪种估计取决于对你最重要的误差种类。 就敌机而言，低估总数可能会造成严重的后果，在这种情况下，你可能会选择使用更加可变的方法，它一半几率都是高估的。 另一方面，如果高估导致了防范不存在的飞机的不必要的高成本，那么你可能会对低估的方法感到满意。

技术注解

事实上，“两倍均值”不是无偏的。平均而言，它正好高估了 1。例如，如果`N`等于 3，来自`1,2,3`的抽取结果的均值是`2`，`2 x 2 = 4`，它比`N`多了 1。“两倍均值”减 1 是`N`的无偏估计量。

十、假设检验

> 原文：Testing Hypotheses
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

数据科学家们经常面对世界的是或不是的问题。你在这个课程中看到了一些这样的问题的例子：

- 巧克力对你有好处吗？
- Broad Street 水泵的水是否会导致霍乱？
- 加州的人口统计在过去的十年中有所改变吗？

我们是否回答这些问题取决于我们的数据。加州的人口普查数据可以解决人口统计的问题，而答案几乎没有任何不确定性。我们知道 Broad Street 水泵的水源受到霍乱病人的污染，所以我们可以很好地猜测它是否会引起霍乱。

巧克力还是其他任何实验对你有好处，几乎肯定要由医学专家来决定，但是第一步是使用数据科学分析来自研究和随机实验的数据。

在本章中，我们将试图回答这样的问题，根据样本和经验分布的结论。我们将以北加利福尼亚州公民自由联盟（ACLU）2010 年进行的一项研究为例。

陪审团选拔

2010 年，ACLU 在加利福尼亚州阿拉米达县提交了一份陪审团选择的报告。报告得出的结论是，在阿拉米达县的陪审团小组成员中，某些族裔人数不足，并建议对专家组进行一些改革，来合理分配陪审员。在本节中，我们将自己分析数据，并检查出现的一些问题。

陪审团

陪审团是一群被选为准陪审员的人；终审的陪审团是从他们中挑选的。陪审团可以由几十人或几千人组成，具体情况取决于审判情况。根据法律，陪审团应该是审判所在社区的代表。加州“民事诉讼法（California's Code of Civil Procedure）”第 197 条规定：“All persons selected for jury service shall be selected at random, from a source or sources inclusive of a representative cross section of the population of the area served by the court.”

最终的陪审团是通过故意纳入或排除，从陪审团中挑选出来的。法律允许潜在的陪审员出于医疗原因而被免责；双方的律师可以从名单上挑选一些潜在的陪审员进行所谓的“先制性反对（peremptory challenges）”。初审法官可以根据陪审团填写的问卷进行选择；等等。但最初的陪审团似乎是合格陪审员的总体的随机样本。

阿拉米达县的陪审团构成

ACLU 的研究重点是阿拉米达县陪审团的种族组成。 ACLU 编辑了 2009 年和 2010 年在阿拉米达县进行的 11 次重罪审判中陪审团的种族组成的数据。在这些陪审团中，报告出庭的陪审员的总人数是 1453 人。ACLU 收集了所有人口的统计数据，并将这些数据与该县所有合格陪审员的组成进行比较。

数据列在下面的表格中，称为 `jury` 。 对于每个种族来说，第一个值就是该种族所有合格的陪审员候选人的比例。 第二个值是出现在出现在陪审团选拔过程的人中，那个种族的人的比例。

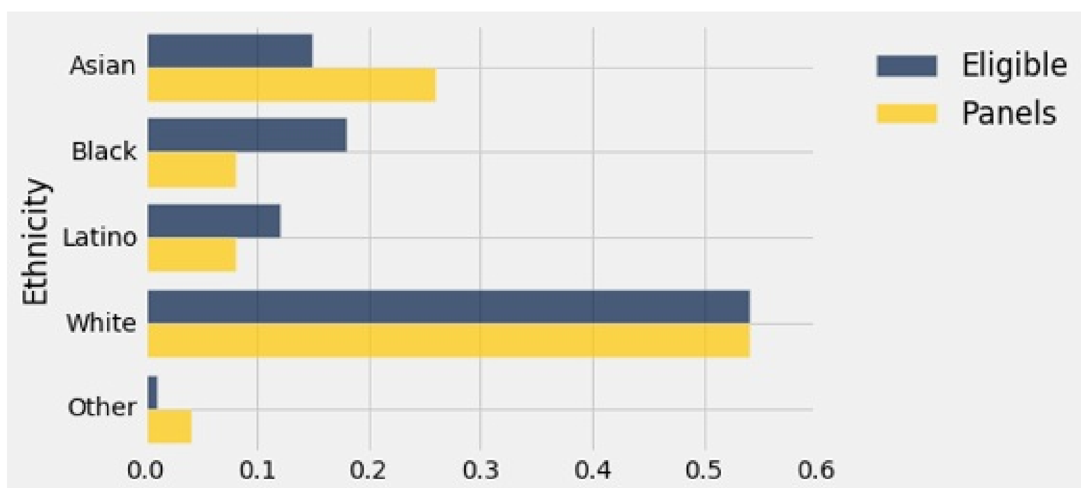
```
jury = Table().with_columns(
    'Ethnicity', make_array('Asian', 'Black', 'Latino', 'White', 'Other'),
    'Eligible', make_array(0.15, 0.18, 0.12, 0.54, 0.01),
    'Panels', make_array(0.26, 0.08, 0.08, 0.54, 0.04)
)

jury
```

Ethnicity	Eligible	Panels
Asian	0.15	0.26
Black	0.18	0.08
Latino	0.12	0.08
White	0.54	0.54
Other	0.01	0.04

研究中的一些种族代表性过多，一些代表性不足。 条形图有助于显示差异。

```
jury.barh('Ethnicity')
```



两个分布的距离

可视化使我们能够快速了解，两个分布之间的相似性和差异。 为了更准确地说出这些差异，我们必须首先量化两个分布之间的差异。 这将使我们的分析能够基于更多东西，不仅仅是我们能够通过眼睛做出的评估。

为了测量两个分布之间的差异，我们将计算一个数量，称之为它们之间的总变异距离（total variation distance）。

为了计算总变异距离，我们首先考虑每个类别中两个比例之间的差异。

```
# Augment the table with a column of differences between proportions

jury_with_diffs = jury.with_column(
    'Difference', jury.column('Panels') - jury.column('Eligible')
)
jury_with_diffs
```

Ethnicity	Eligible	Panels	Difference
Asian	0.15	0.26	0.11
Black	0.18	0.08	-0.1
Latino	0.12	0.08	-0.04
White	0.54	0.54	0
Other	0.01	0.04	0.03

```
jury_with_diffs.column('Abs. Difference').sum()/2
0.14000000000000001
```

这个数量 0.14 是合格陪审员总体中种族分布与陪审团分布情况之间的总变异距离（TVD）。

只要加上正的差异，我们就可以得到相同的结果。 但是，我们的方法包含所有绝对差异，不需要追踪哪些差异是正的而哪些不是。

计算 TVD 的函数

函数 `total_variation_distance` 返回两个数组中的分布的 TVD。

```
def total_variation_distance(distribution_1, distribution_2):
    return np.abs(distribution_1 - distribution_2).sum()/2
```

函数 `table_tvd` 使用函数 `total_variation_distance` 来返回表的两列中的分布的 TVD。

```
def table_tvd(table, label, other):
    return total_variation_distance(table.column(label), table.column(other))

table_tvd(jury, 'Eligible', 'Panels')
0.14000000000000001
```

陪审团是否是总体的代表？

现在我们将转到合格的陪审员和陪审团的 TVD 的值。我们如何解释 0.14 的距离呢？要回答这个问题，请回想一下，陪审团应该是随机选择的。因此，将 0.14 的值与合格的陪审员和随机选择的陪审团的 TVD 进行比较，会有帮助。

为了这样做，我们将在模拟中使用我们的技能。研究共有 1453 名准陪审员 。所以让我们从合格的陪审员的总体中随机抽取大小为 1453 的样本。

技术注解。准陪审员的随机样本将会不放回地选中。但是，如果样本的大小相对于总体的大小较小，那么无放回的取样类似于放回的取样；总体中的比例在几次抽取之间变化不大。阿拉米达县的合格陪审员的总体超过一百万，与此相比，约 1500 人的样本量相当小。因此，我们将带放回地抽样。

从合格的陪审员中随机抽样

到目前为止，我们已经使用 `np.random.choice` 从数组元素中随机抽样，并使用 `sample` 对表的行进行抽样。 但是现在我们必须从一个分布中抽样：一组种族以及它们的比例。

为此，我们使用函数 `proportions_from_distribution` 。 它有三个参数：

- 表名
- 包含比例的列的标签
- 样本大小

该函数执行带放回地随机抽样，并返回一个新的表，该表多出了一列 `Random Sample` ，是随机样本中所出现的比例。

所有陪审团的总大小是 1453，所以让我们把这个数字赋给给一个名成，然后调用：

```
proportions_from_distribution.

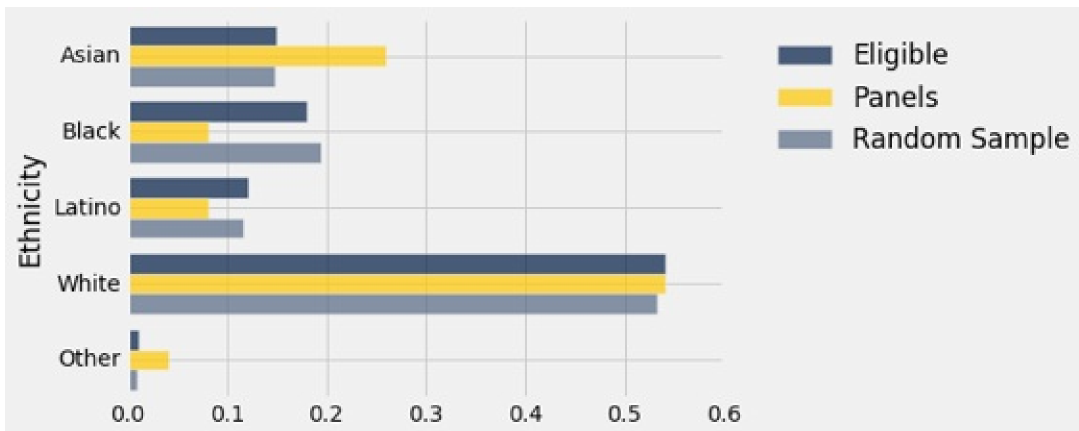
panel_size = 1453
panels_and_sample = proportions_from_distribution(jury, 'Eligible', panel_size)
panels_and_sample
```

Ethnicity	Eligible	Panels	Random Sample
Asian	0.15	0.26	0.14797
Black	0.18	0.08	0.193393
Latino	0.12	0.08	0.116311
White	0.54	0.54	0.532691
Other	0.01	0.04	0.00963524

从结果中可以清楚地看出，随机样本的分布与合格总体的分布非常接近，与陪审团的分布不同。

和之前一样，可视化会有帮助。

```
panels_and_sample.barh('Ethnicity')
```



灰色条形与蓝色条形比金色条形更接近。 随机样本类似于合格的总体，而不是陪审团。

我们可以通过计算合格总体的分布与随机样本之间的 TVD，来量化这一观察结果。

```
table_tvd(panels_and_sample, 'Eligible', 'Random Sample')
0.013392980041293877
```

将其与陪审团的距离 0.14 进行比较，可以看到我们在条形图中看到的数值。 合格总体与陪审团之间的 TVD 为 0.14，但合格总体与随机样本之间的 TVD 小得多。

当然，随机样本和合格陪审员的分布之间的距离取决于样本。 再次抽样可能会给出不同的结果。

随机样本和总体之间有多少差异？

随机样本与合格陪审员的分布之间的 TVD，是我们用来衡量两个分布之间距离的统计量。 通过重复抽样过程，我们可以看到不同随机样本的统计量是多少。 下面的代码根据抽样过程的大量重复，来计算统计量的经验分布。

```
# Compute empirical distribution of TVDs

panel_size = 1453
repetitions = 5000

tvds = make_array()

for i in np.arange(repetitions):

    new_sample = proportions_from_distribution(jury, 'Eligible', panel_size)
    tvds = np.append(tvds, table_tvd(new_sample, 'Eligible', 'Random Sample'))

results = Table().with_column('TVD', tvds)
results
```

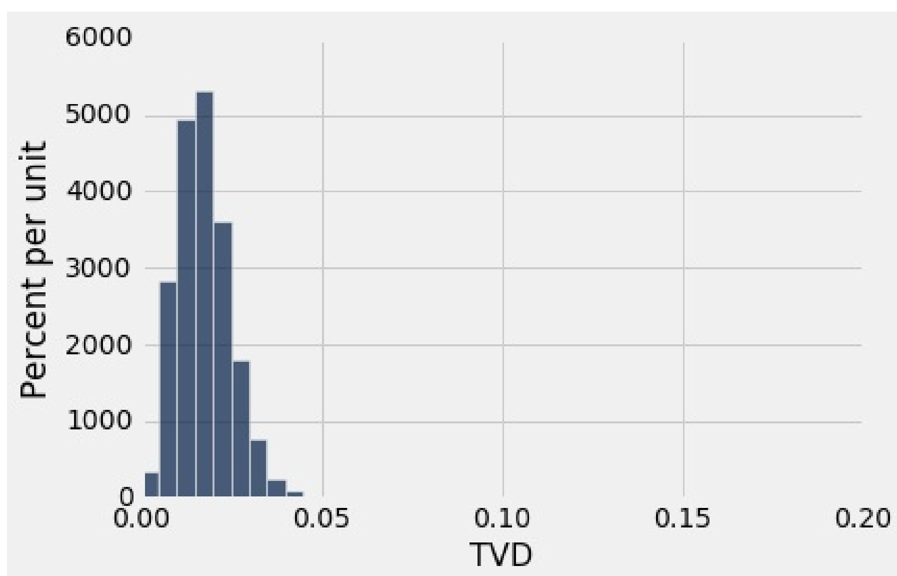
TVD
0.0247075
0.0141569
0.0138403
0.0214384
0.012278
0.017309
0.0219752
0.0192017
0.02351
0.00818995

（省略了 4990 行）

上面每一行包含大小为 1453 的随机样本与合格的陪审员的 TVD。

这一列的直方图显示，从合格候选人中随机抽取 1453 名陪审员的结果是，偏离合格陪审员的种族分布的分布几乎不超过 0.05。

```
results.hist(bins=np.arange(0, 0.2, 0.005))
```



陪审团和随机样本比如何？

然而，研究中的陪审团与合格总体并不十分相似。陪审团和总体之间的 TVD 是 0.14，这距离上面的直方图的尾部很远。这看起来不像是随机样本和合格总体之间的典型距离。

所以我们的分析支持 ACLU 的计算，即陪审团不是合格陪审员的分布的代表。然而，与大多数这样的分析一样，它并没有说明分布为什么不同，或者差异可能暗示了什么。

ACLU 报告讨论了这些差异的几个可能的原因。例如，一些少数群体在选民登记记录和机动车辆部门（选择陪审员的两个主要来源）的代表性不足。在进行研究时，该县没有一个有效的程序，用于跟踪那些被选中但未出庭的准陪审员。ACLU 列举了其他几个原因。不管出于何种原因，似乎很明显，陪审团的组成与我们对随机样本的预期不同，它来自 `Eligible` 列的分布。

数据上的问题

我们已经开发出一种强大的技术，来帮助决定一个分布是否像另一个分布的随机样本。但是数据科学不仅仅是技术。特别是数据科学总是需要仔细研究如何收集数据。

合格的陪审员。首先，重要的是要记住，不是每个人都有资格担任陪审团的职位。阿拉米达县高级法院在其网站上说："如果你是18 岁的美国公民，和传召所在的县或区的居民，你可能会被要求担任职位。你必须能够理解英语，身体上和精神上都有能力担任，此外，你在过去 12 个月内不得担任任何类型的陪审员，也没有被判重罪。

人口普查没有保存所有这些类别的人口记录。因此 ACLU 必须以其他方式获得合格陪审员的人口统计资料。以下是他们对自己所遵循的过程的描述，以及它可能包含的一些缺陷。

"为了确定阿拉米达县具有陪审团资格的人口的统计数据，我们使用了一个声明，它为阿拉米达县人民起诉斯图亚特·亚历山大的审判而准备。在声明中，圣地亚哥州立大学的人口统计学家 Weeks 教授，根据 2000 年的人口普查数据估算了阿拉米达县的具有陪审团资格的人口，

为了得出这个估计值，Weeks 教授考虑到了不符合陪审团担任条件的人数，因为他们不会说英文，不是公民，因此，他的估计应该是对阿拉米达县实际具有陪审团资格的人口的准确评估，而不仅仅是审查居住在阿拉米达的所有人口的种族和族裔的人口普查报告。应该指出的是，Weeks 教授所依据的人口普查数据现在已经有十年了，县的人口统计数据的某些类别，可能已经改变了两到三个百分点。”

因此，分析中使用的合格陪审员的种族分布本身就是一个估计，可能有点过时。

陪审团。 此外，陪审团并不从整个合格总体中选出。 阿拉米达县高等法院说：“法院的目标是提供县人口的准确的横截面，陪审员的名字是从登记选民和/或车管局发出的驾驶执照中随机抽取的”。

所有这些都产生了复杂问题，就是如何准确估计阿拉米达县合格陪审员的种族构成。

目前还不清楚，1453 个陪审团成员如何划分为不同的种族类别（ACLU 报告称“律师......合作收集陪审团数据”）。 存在严重的社会，文化和政治因素，影响谁被归类或自我分类到每个种族类别。 我们也不知道陪审团中这些类别的定义，是否与 Weeks 教授所使用的定义相同，Weeks 教授又在它的估算过程中使用了人口普查类别。 因此被比较的两个分布的对应关系，也存在问题。

美国最高法院，1965年：斯温 VS 阿拉巴马州

在二十世纪六十年代初期，阿拉巴马州的塔拉迪加县，一个名叫罗伯特·斯温的黑人被指控强奸一名白人妇女，并被判处死刑。 他援引所有陪审团是白人的其他因素，对他的判决提出上诉。当时，只有 21 岁或以上的男子被允许在塔拉迪加县的陪审团中任职。 在县里，合格的陪审员中有 26% 是黑人，但在 Swain 的审判中选出的 100 名陪审团中只有 8 名黑人男子。审判陪审团没有选定黑人。

1965 年，美国最高法院驳回了斯温的上诉。 法院在其裁决中写道：“整体百分比差距很小，没有反映出包括或排除特定数量的黑人的尝试”。（... the overall percentage disparity has been small and reflects no studied attempt to include or exclude a specified number of Negroes.）

让我们用我们开发的方法来检查，陪审团中的 100 名黑人中的 8 名与合格陪审员的分布之间的差异。

```
swain_jury = Table().with_columns(
    'Ethnicity', make_array('Black', 'Other'),
    'Eligible', make_array(0.26, 0.74),
    'Panel', make_array(0.08, 0.92)
)

swain_jury
```

Ethnicity	Eligible	Panel
Black	0.26	0.08
Other	0.74	0.92

```
table_tvd(swain_jury, 'Eligible', 'Panel')
0.18000000000000002
```

两个分布之间的 TVD 是 0.18。 这与合格总体的分布和随机样本之间的 TVD 比较如何？

为了回答这个问题，我们可以模拟从随机样本中计算的 TVD。

```
# Compute empirical distribution of TVDs

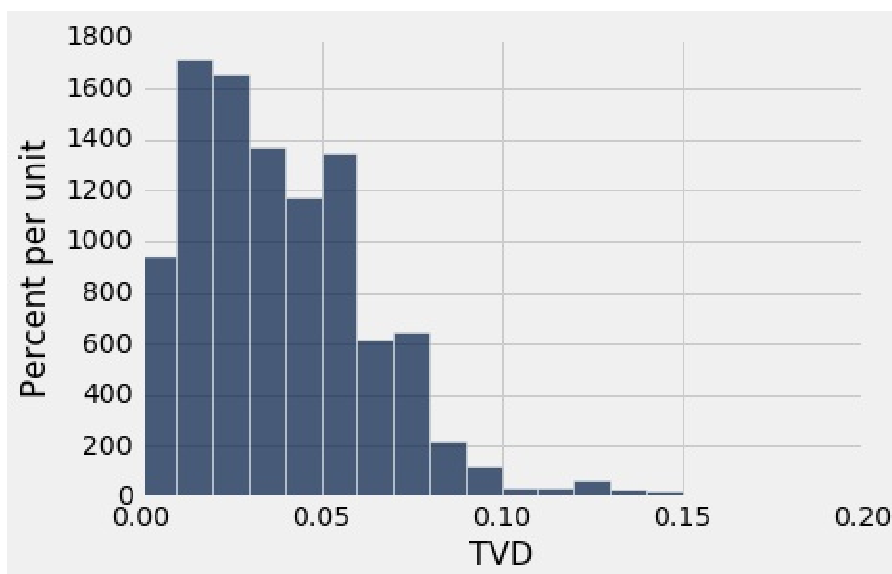
panel_size = 100
repetitions = 5000

tvds = make_array()

for i in np.arange(repetitions):

    new_sample = proportions_from_distribution(swain_jury, 'Eligible', panel_size)
    tvds = np.append(tvds, table_tvd(new_sample, 'Eligible', 'Random Sample'))

results = Table().with_column('TVD', tvds)
results.hist(bins = np.arange(0, 0.2, 0.01))
```



随机样本的 TVD 小于我们所得的值 0.18，它是陪审团和合格陪审员的 TVD。

在这个分析中，数据并没有像我们以前的分析那样被问题盖住 - 涉及的人总数相对较少，而且最高法院案件的统计工作也很仔细。

因此，我们的分析有了明确的结论，那就是陪审团不是总体的代表。 最高法院的判决“整体百分比差距很小”是很难接受的。

检验的术语

在陪审团选择的例子的背景下，我们已经形成了一些假设统计检验的基本概念。使用统计检验作为决策的一种方法是许多领域的标准，并且存在标准的术语。以下是大多数统计检验中的步骤顺序，以及一些术语和示例。

第一步：假设

所有的统计检验都试图在世界的两种观点中进行选择。具体而言，选择是如何生成数据的两种观点之间的选择。这两种观点被称为假设。

原（零）假设。这就是说，数据在明确指定的假设条件下随机生成，这些假设使计算几率成为可能。“零”一词强化了这样一个观点，即如果数据看起来与零假设的预测不同，那么这种差异只是偶然的。

在阿拉米达县陪审团选择的例子中，原假设是从合格的陪审员人群中，随机抽取这些陪审团。虽然审团的种族组成与合格的陪审员的总体不同，但除了机会变异以外，没有任何理由存在差异。

备选假设。这就是说，除了几率以外的某些原因使数据与原假设所预测的数据不同。非正式而言，备选假设认为观察到的差异是“真实的”。

在我们阿拉米达县陪审团选择的例子中，备选假设是，这些小组不是随机选出来的。除了几率以外的事情导致了，陪审团的种族组成和合格陪审员总体的种族组成之间存在差异。

第二步：检验统计量

为了在这两个假设之间作出决策，我们必须选择一个统计量作为我们决策的依据。 这被称为检验统计量。

在阿拉米达县陪审团的例子中，我们使用的检验统计量是，陪审团与合格陪审员的总体的种族分布之间的总变异距离。

计算检验统计量的观察值通常是统计检验中的第一个计算步骤。 在我们的例子中，陪审团与总体之间的总变异距离的观察值是 0.14。

第三步：检验统计量的概率分布，在原假设下

这个步骤把检验统计量的观察值放在一边，而是把重点放在，如果原假设为真，统计量的值是什么。 在原假设下，由于几率，样本可能出现不同的情况。 所以检验统计量可能会有所不同。 这个步骤包括在随机性的原假设下，计算出所有可能的检验统计量及其所有概率。

换句话说，在这个步骤中，我们假设原假设为真，并计算检验统计量的概率分布。 对于许多检验统计量来说，这在数学和计算上都是一项艰巨的任务。 因此，我们通过抽样过程的大量重复，通过统计量的经验分布来近似检验统计量的概率分布。

在我们的例子中，我们通过直方图可视化了这个分布。

第四步 检验的结论

原假设和备选假设之间的选择，取决于步骤 2 和 3 的结果之间的比较：检验统计量的观察值以及它的分布，就像由原假设预测的那样。

如果二者一致，则观察到的检验统计量与原假设的预测一致。 换句话说，这个检验并不偏向备选假设；数据更加支持原假设。

但如果两者不一致，就像我们阿拉米达县陪审团的例子那样，那么数据就不支持原假设。 这就是为什么我们得出结论，陪审团不是随机挑选的。 几率之外的东西影响了他们的构成。

如果数据不支持原假设，我们说检验拒绝了原假设。

孟德尔的豌豆花

格雷戈·孟德尔（1822-1884）是一位奥地利僧侣，被公认为现代遗传学领域的奠基人。 孟德尔对植物进行了仔细而大规模的实验，提出遗传学的基本规律。

他的许多实验都在各种豌豆上进行。 他提出了一系列每个品种的假设。 这些被称为模型。 然后他通过种植植物和收集数据来测试他的模型的有效性。

让我们分析这样的实验的数据，看看孟德尔的模型是否好。

在一个特定的品种中，每个植物具有紫色或白色的花。 每个植物的颜色不受其他植物颜色的影响。 孟德尔推测，植物应随机具有紫色或白色的花，比例为 3：1。

原假设。 对于每种植物，75% 的几率是紫色的花，25% 的几率是白色的花，无论其他植物的颜色如何。

也就是说，原假设是孟德尔的模型是好的。 任何观察到的模型偏差都是机会变异的结果。

当然，有一个相反的观点。

备选假设。 孟德尔的模型是无效的。

让我们看看孟德尔收集的数据更加支持这些假设中的哪一个。

`flowers` 表包含了由模型预测的比例，以及孟德尔种植的植物数据。

```
flowers = Table().with_columns(
    'Color', make_array('Purple', 'White'),
    'Model Proportion', make_array(0.75, 0.25),
    'Plants', make_array(705, 224)
)

flowers
```

Color	Model Proportion	Plants
Purple	0.75	705
White	0.25	224

共有 929 株植物。 为了观察颜色的分布是否接近模型预测的结果，我们可以找到观察到的比例和模型比例之间的总变异距离，就像我们之前那样。 但是只有两个类别（紫色和白色），我们有一个更简单的选择：我们可以查看紫色的花的比例。 白色的比例没有新的信息，因为它只是 1 减去紫色的比例。

```
total_plants = flowers.column('Plants').sum()
total_plants
929
observed_proportion = flowers.column('Plants').item(0)/total_plants
observed_proportion
0.7588805166846071
```

检验统计量。 由于该模型预测 75% 的植物花为紫色，相关的统计量是 0.75 与观察到的花为紫色的植物的比例之间的差异。

```
observed_statistic = abs(observed_proportion - 0.75)
observed_statistic
0.0088805166846070982
```

这个值与原假设所说的应该的情况相比如何？ 为了回答这个问题，我们需要使用模型来模拟植物的新样本并计算每个样本的统计量。

我们将首先创建数组`model_colors`，包含颜色，比例由模型给定。 然后我们可以使用`np.random.choice`从这个数组中，带放回地随机抽样 929 次。 根据孟德尔的模型，这就是植物的生成过程。

```
model_colors = make_array('Purple', 'Purple', 'Purple', 'White')
new_sample = np.random.choice(model_colors, total_plants)
```

> 译者注：这里可以使用`np.random.choice`的`p`参数来简化编程。
>
> ```
> new_sample = np.random.choice(['Purple', 'White'], total_plants, p=[0.75, 0.25])
> ```

为了与我们观察到的统计量进行比较，我们需要知道这个新样本中，花为紫色的植物的比例与 0.75 的差。

```
proportion_purple = np.count_nonzero(new_sample == 'Purple')/total_plants
abs(proportion_purple - 0.75)
0.016953713670613602
```

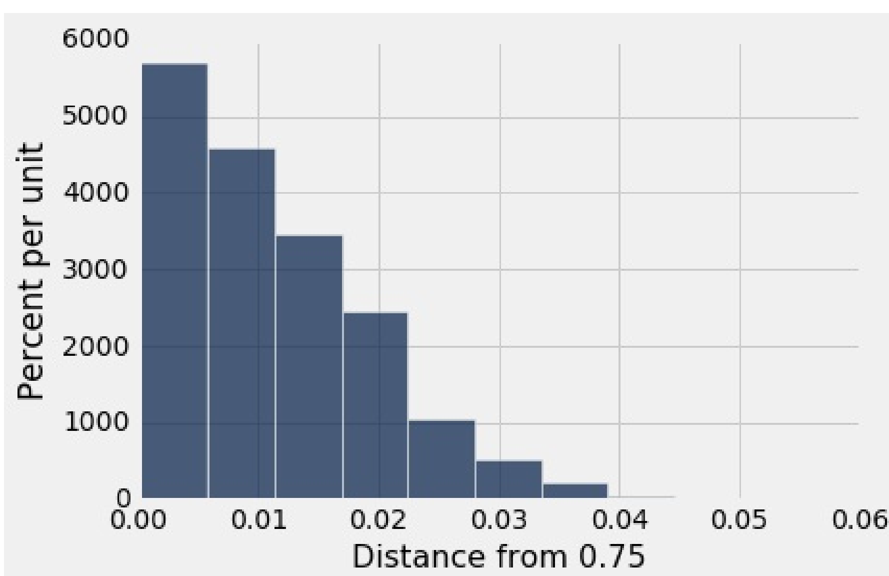
检验统计量的经验分布，在原假设为真的情况下。 毫不奇怪，我们得到的值与我们观察到的统计量之间的差约为 0.00888。 但是如果我们又取了一个样本，会有多大的不同呢？ 你可以通过重新运行上面的两个单元格来回答这个问题，或者使用 `for` 循环来模拟统计量。

```
repetitions = 5000

sampled_stats = make_array()

for i in np.arange(repetitions):
    new_sample = np.random.choice(model_colors, total_plants)
    proportion_purple = np.count_nonzero(new_sample == 'Purple')/total_plants
    sampled_stats = np.append(sampled_stats, abs(proportion_purple - 0.75))

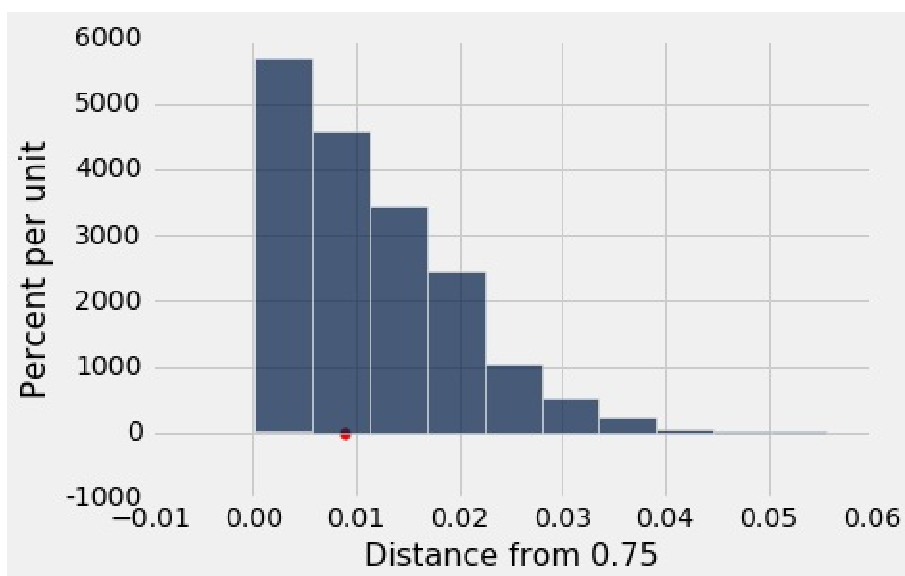
results = Table().with_column('Distance from 0.75', sampled_stats)
results.hist()
```



检验的结论。 根据孟德尔的数据，统计量的观测值是 0.00888，刚好 0.01 以下。 这正好在这个分布的中心。

```
results.hist()

#Plot the observed statistic as a large red point on the horizontal axis
plots.scatter(observed_statistic, 0, color='red', s=30);
```



基于孟德尔数据的统计量，与我们基于孟德尔模型的模拟的分布是一致的。 因此，与备选假设相比，数据更加支持原假设 - 孟德尔的模型是好的。

P 值和“一致”的含义

在阿拉米达县陪审团的例子中，我们观察到的检验统计量显然与原假设的预测差距很大。在豌豆花的例子中，观察到的统计量与原假设所预测的分布一致。所以在这两个例子中，选择哪个假设是明显的。

但是有时候这个决策还不是很明显。观察到的检验统计量是否与原假设预测的分布一致，是一个判断问题。我们建议你使用检验统计量的值以及原假设预测的分布图，来做出判断。这将使你的读者可以自己判断两者是否一致。

如果你不想做出自己的判断，你可以遵循一些惯例。这些惯例基于所谓的观察到的显着性水平，或简称 P 值。 P 值是一个几率，使用检验统计量的概率分布计算，可以用步骤 3 中的经验分布来近似。

求出 P 值的实用说明。现在，我们只是给出一个求出该值的机械的方法；意义和解释放到下一节中。方法：将观察到的检验统计量放在直方图的横轴上，求出从以该点起始的尾部比例。这就是 P 值，或者是基于经验分布的 P 值的相当好的近似值。

```
empirical_P = np.count_nonzero(sampled_stats >= observed_statistic)/repetitions
empirical_P
0.5508
```

观察到的统计量 0.00888 非常接近孟德尔模型下所有统计量的中位数。 你可以把它看作是我们之前评论的一个量化，即观察到的统计量正好在原假设的分布中心。

但是如果离得更远呢？ 例如，如果观察到的统计量是 0.035 呢？ 那么我们会得出什么结论呢？

```
np.count_nonzero(sampled_stats >= 0.035)/repetitions
0.0122
```

这个比例就很小了。 如果 P 值较小，那就意味着它的尾部很小，所以观察到的统计量远离原假设的预测。 这意味着数据支持备选假设而不是支持原假设。

所以如果我们观察到的统计量是 0.035 而不是 0.00888，我们会选择备选假设。

那么多小算“小”呢？ 这里有个约定。

- 如果 P 值小于 5%，结果称为“统计学显着”。
- 如果 P 值更小 - 小于 1%，结果被称为“高度统计学显着”。

在这两种情况下，检验的结论是数据支持备选假设。

约定的历史注解

上面定义的统计学显着性的确定，已经在所有应用领域的统计分析中成为标准。当一个约定被如此普遍遵循时，研究它是如何产生的就有趣了。

统计检验方法 - 基于随机样本数据在假设之间选择 - 由 Ronald Fisher 爵士在 20 世纪初开发。在 1925 年出版的《写给研究工作者的统计学方法》（Statistical Methods for Research Workers）一书中的下列陈述中，Ronald 爵士可能在不知情的情况下建立了统计学显著的约定。对于 5% 的水平，他写道：“判断一个偏差是否显著的时候，将它当做一个极限非常方便。

Ronald 爵士觉得“方便”的东西变成了截断，获得了普适常数的地位。无论罗纳德爵士如何选出了这个点，这个值是他在众多值中的个人选择：在 1926 年的一篇文章中，他写道：“如果二十分之一看起来还是不够高，如果我们愿意的话， 我们可以把线画在百分之二的地方，或者百分之一。个人来说，作者更倾向于把显著的较低标准设为 5%...”

Fisher 知道“低”是一个判断问题，没有独特的定义。我们建议你遵循他的优秀例子。提供你的数据，作出判断，并解释你为什么这样做。

GSI 的辩护

假设检验是最广泛使用的统计推断方法之一。我们已经看到，它的用途十分广泛，例如审团选择和豌豆花。在本节的最后一个例子中，我们将在另一个完全不同的语境中对假设进行测试。

伯克利统计班的 350 名学生被分为 12 个讨论小组，由研究生导师（GSI）带领。期中之后后，第三组的学生注意到，他们的成绩平均上低于班上的其他人。

在这种情况下，学生们往往会抱怨这一组的 GSI 。他们肯定觉得，GSI 的教学一定是有问题的。否则为什么他们组会比别人做得更差呢？

GSI 通常有更多的统计学经验，他们的观点往往是不同的：如果你只是从全班随机抽取一部分学生，他们的平均分数就可能与学生不满意的分数相似。

GSI 的立场是一个明确的几率模型。我们来检验一下。

原假设：第三组的平均成绩类似于从班上随机抽取的相同数量的学生的平均成绩。

备选假设：不是，太低了。

`scores` 包含整个班级的每个学生的小组编号和期中成绩。期中成绩是 0 到 25 的整数；0 的意思是学生没来考试。

```
scores = Table.read_table('scores_by_section.csv')
scores
```

Section	Midterm
1	22
2	12
2	23
2	14
1	20
3	25
4	19
1	24
5	8
6	14

（省略了 349 行）

这是 12 个小组的平均成绩。

```
scores.group('Section', np.mean).show()
```

Section	Midterm mean
1	15.5938
2	15.125
3	13.6667
4	14.7667
5	17.4545
6	15.0312
7	16.625
8	16.3103
9	14.5667
10	15.2353
11	15.8077
12	15.7333

第三组平均成绩比其他组低一点。 这看起来像机会变异？

我们知道如何找出答案。 我们首先从全班随机挑选一个“第三组”，看看它的平均得分是多少；然后再做一遍又一遍。

首先，我们需要第三组的学生人数：

```
scores.group('Section')
```

Section	count
1	32
2	32
3	27
4	30
5	33
6	32
7	24
8	29
9	30
10	34

（省略了 2 行）

现在我们的计划是，从班上随机挑选 27 名学生，并计算他们的平均分数。

所有学生的成绩都在一张表上，每个学生一行。 因此，我们将使用 `sample` 来随机选择行，使用 `with_replacement = False` 选项，以便我们无放回地抽样。（稍后我们会看到，结果几乎与我们通过放回取样所得到的结果相同）。

```
scores.sample(27, with_replacement=False).column('Midterm').mean()
13.703703703703704
```

我们已经准备好，模拟随机的“第三组”的均值的经验分布。

```
section_3_mean = 13.6667

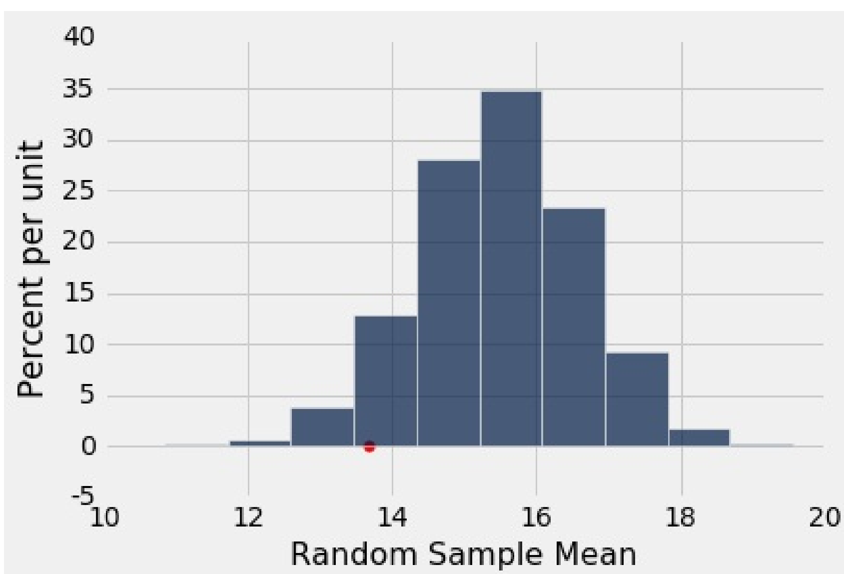
repetitions = 10000

means = make_array()

for i in np.arange(repetitions):
    new_mean = scores.sample(27, with_replacement=False).column('Midterm').mean()
    means = np.append(means, new_mean)

emp_p_value = np.count_nonzero(means <= section_3_mean)/repetitions
print('Empirical P-value:', emp_p_value)
results = Table().with_column('Random Sample Mean', means)
results.hist()

#Plot the observed statistic as a large red point on the horizontal axis
plots.scatter(section_3_mean, 0, color='red', s=30);
Empirical P-value: 0.0581
```



从直方图来看，第三组的较低均值看起来有些不寻常，但 5% 截断值的惯例更加偏向 GSI 的假设。 有了这个截断值，我们说这个结果不是统计学显著的。

错误概率

在我们决定我们的数据更加支持哪个假设的过程中，最后一步涉及数据的原假设的一致性判断。 虽然绝大多数时候这一步都能产生正确的决策，但有时也会让我们误入歧途。 原因是机会变异。 例如，即使当原假设为真时，机会变异也可能导致样本看起来与原假设的预测完全不同。

在本节中，我们将研究假设的统计检验如何可能得出这样的结论，也就是实际上原假设为真时，数据支持备选假设。

由于我们根据 P 值做出决策，现在应该给出一个更正式的定义，而不是“在经验直方图的横坐标上放置观察到的统计量，并且求出大于它的尾部区域”的机械方法。

P 值的定义

P 值是在原假设下，检验统计量等于在数据中观察到的值，或甚至在备选假设方向上更进一步的几率。

让我们先看看这个定义如何与前一节的计算结果一致。

回顾孟德尔的豌豆花

在这个例子中，我们评估孟德尔的豌豆物种的遗传模型是否良好。 首先回顾一下我们如何建立决策过程，然后在这个背景下考察 P 值的定义。

原假设。 孟德尔的模型是好的：植物的花是紫色或白色，类似于来自总体紫色，紫色，紫色，白色的带放回随机样本。

备选假设。 孟德尔的模型是错误的。

检验统计量。0.75 与花为紫色的植物的观察比例的距离：

$$\text{test statistic} ~=~ |\text{observed proportion purple} - 0.75|$$

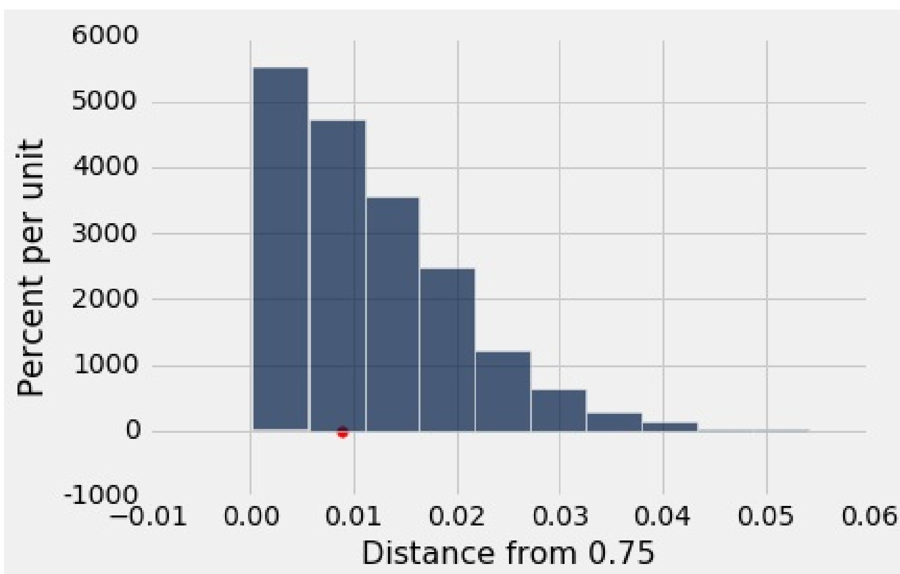
样本量较大（929），所以如果孟德尔的模型好，那么观察到的紫色花的比例应该接近 0.75。如果孟德尔的模型是错误的，则观察到的紫色比例不应该接近0.75，从而使统计值量更大。

因此，在这种情况下，“备选假设的方向”意味着“更大”。 检验统计量的观测值（四舍五入到小数点后五位）是$|0.75888 - 0.75| ~=~ 0.00888$。根据定义，P 值是从孟德尔的模型中抽取的样本，产生 0.00888 或更大的统计量的几率。

虽然我们还没有学会如何精确地计算这个几率，但我们可以通过模拟来逼近它，这就是我们在前一节中所做的。 以下是该部分的所有相关代码。

```
# The model and the data
model_colors = make_array('Purple', 'Purple', 'Purple', 'White')
total_plants = 929
observed_statistic = 0.0088805166846070982
# Simulating the test statistic under the null hypothesis
repetitions = 5000
sampled_stats = make_array()
for i in np.arange(repetitions):
    new_sample = np.random.choice(model_colors, total_plants)
    proportion_purple = np.count_nonzero(new_sample == 'Purple')/total_plants
    sampled_stats = np.append(sampled_stats, abs(proportion_purple - 0.75))

# The P-value (an approximation based on the simulation)
empirical_P = np.count_nonzero(sampled_stats >= observed_statistic)/repetitions
# Displaying the results
results = Table().with_column('Distance from 0.75', sampled_stats)
print('Empirical P-value:', empirical_P)
results.hist()
plots.scatter(observed_statistic, 0, color='red', s=30);
Empirical P-value: 0.5436
```



注意 P 值的计算根据孟德尔的模型，基于所有抽取样本的重复，并且每次都计算检验统计量：

```
empirical_P = np.count_nonzero(sampled_stats >= observed_statistic)/repetitions
empirical_P
0.5436
```

这是统计量大于等于观测值 0.00888 的样本比例。

计算结果表明，如果孟德尔的假设是真实的，那么得到一个植物样本，它的检验统计量大于等于孟德尔的观测值，这个几率大概是 54%。 这是一个很大的几率（并且比“较小”的惯例上的 5% 截断值要大得多）。 因此，孟德尔的数据产生了一个统计量，基于他的模型是不足为奇的，这个数据支持他的模型而不是支持备选假设。

回顾 GSI 的辩护

在这个例子中，第三组由一个班级 12 个组中的 27 个学生组成，期中分数均值低于其他组。我们试图在以下假设之间作出决策：

原假设：第三组的平均分数类似于从班上随机挑选的 27 名学生的平均分数。

备选假设：不是，太低了。

检验统计量。 抽样分数的均值。

在这里，备选假设说了，观察到的平均值太低，并不从随机抽样中产生 - 第三组里面有些东西使得平均值较低。

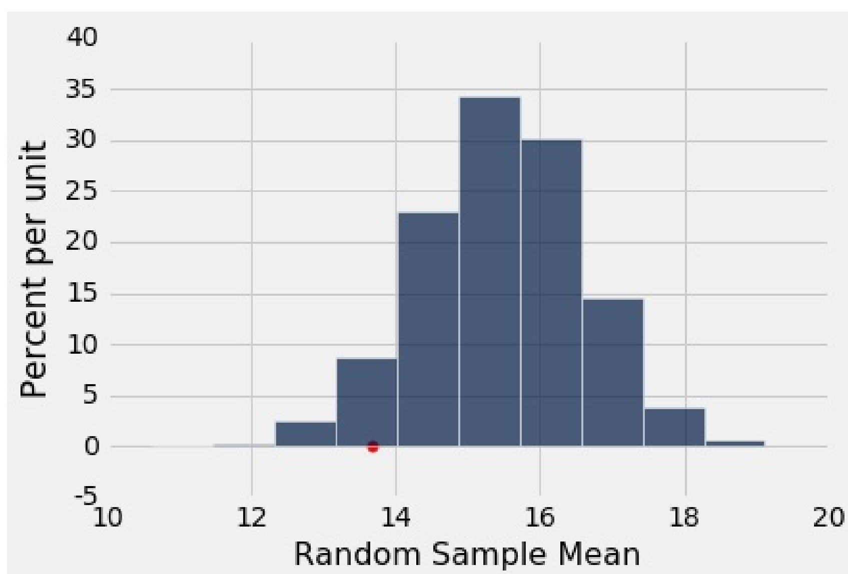
所以在这里，“备选假设的方向”是指“较小”。

检验统计量的观测值是第三组的平均分 13.6667。因此，根据定义，P 值是 27 位随机选取的学生的平均分 13.6667 或更小的几率。

这是我们通过近似来模拟的几率。 这是上一节的代码。

```
# The data
scores = Table.read_table('scores_by_section.csv')
sec_3_mean = 13.6667
sec_3_size = 27
# Simulating the test statistic under the null hypothesis
repetitions = 10000
means = make_array()
for i in np.arange(repetitions):
    new_mean = scores.sample(sec_3_size, with_replacement=False).column('Midterm').mea
n()
    means = np.append(means, new_mean)

# The P-value (an empirical approximation based on the simulation)
empirical_P = np.count_nonzero(means <= sec_3_mean)/repetitions
# Displaying the results
print('Empirical P-value:', empirical_P)
results = Table().with_column('Random Sample Mean', means)
results.hist()
plots.scatter(sec_3_mean, 0, color='red', s=30);
Empirical P-value: 0.0569
```



经验 P 值的计算在下面的单元格中。

```
empirical_P = np.count_nonzero(means <= sec_3_mean)/repetitions
empirical_P
0.0569
```

这是随机样本的比例，其中样本均值小于等于第三组的均值 13.667。

模拟结果显示，随机抽样组的 27 名学生平均分数低于第三组的均值，几率为大约 6% 。如果按照传统的 5% 截断值作为“较小” P 值的定义，那么 6% 不小了，结果不是统计学显着的。换句话说，你没有足够的证据来拒绝原假设的随机性。

你可以尽管违背约定，选择不同的截断值。如果你这样做，请记住以下几点：

- 始终提供检验统计量的观察值和 P 值，以便读者可以自行决定 P 值是否小。
- 只有当传统的所得结果不符合你的喜好时，才需要违背约定。
- 即使你的检验结论为，第三组平均分数低于随机抽样的学生的平均分数，也没有为什么它较低的信息。

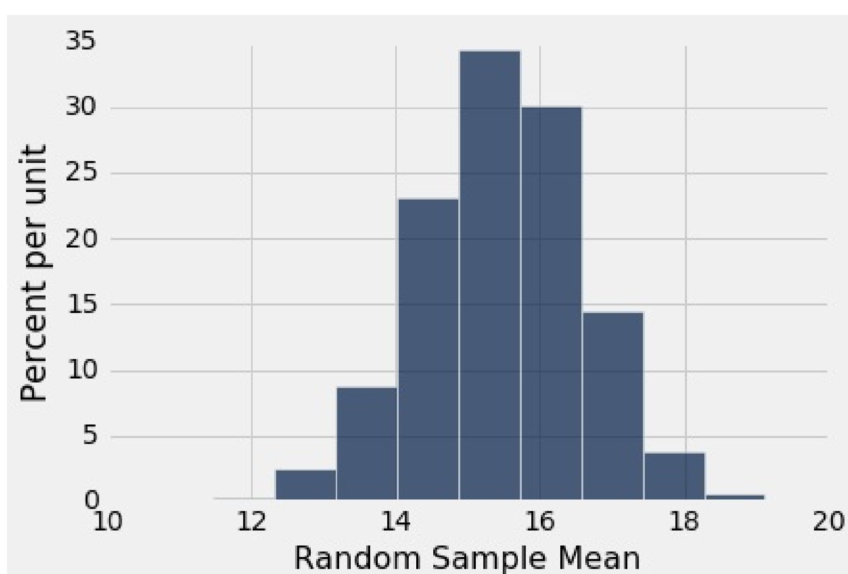
做出错误决策的概率

这种平均分数的分析产生了一个重要的观测，关于我们的检验做出错误结论的概率。

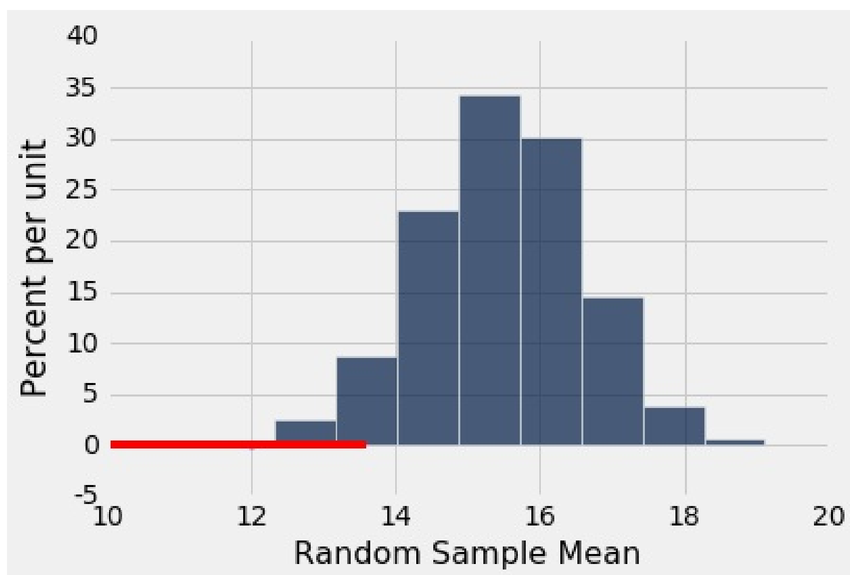
假设你决定使用 5% 的截断值作为 P 值。 也就是说，如果 P 值低于 5%，那么假设你会选择备选假设，否则保持原假设。

那么从样本均值的经验直方图可以看出，如果第三组的平均值是 12，那么你会说“太低了”。12 左侧的面积不足 5%。

```
results.hist()
```



13 左边的面积也不到 5%。 左侧面积小于 5% 的所有样本均值以红色显示。



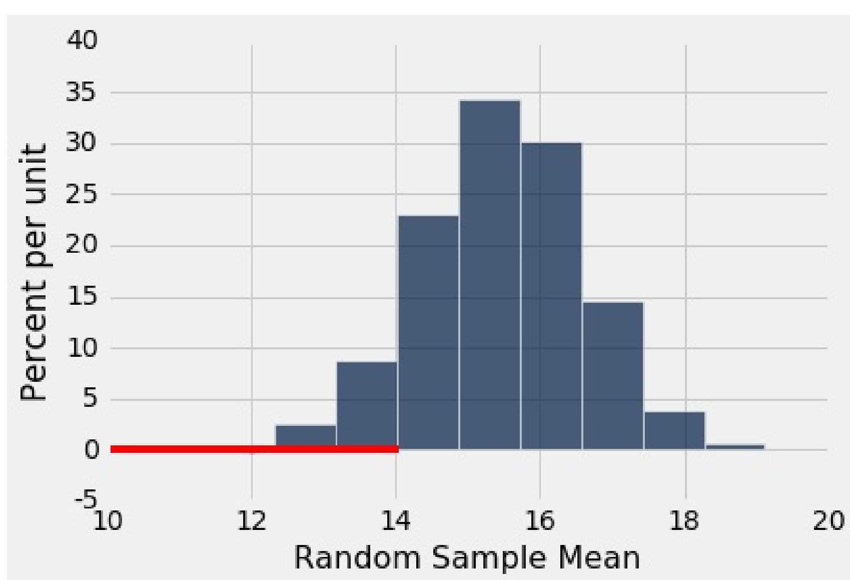
你可以看到，如果第三组的平均值接近 13，并且你使用 5% 的截断值作为 P 值，那么你应该说小组的均值不像随机样本的均值。

你也可以看到，随机样本的均值可能在 13 左右（尽管不太可能）。事实上，在我们的模拟中，5000 个随机样本中有几个的均值与 13 相差 0.01 以内。

```
results.where('Random Sample Mean', are.between(12.99, 13.01)).num_rows
13
```

你看到的是检验做出错误结论的可能性。

如果你使用了 10% 的截断值而不是 5%，那么这里的红色部分意味着，你可能得出结论，它太低了，不能从随机样本中产生，即使在你不知情的情况下，它们是来自随机样本。



做出错误决策的几率

假设你想测试一个硬币是否均匀。 那么假设是：

原假设：硬币是均匀的。 也就是说，结果是来自正面和反面的随机样本。

备选假设：硬币不均匀。

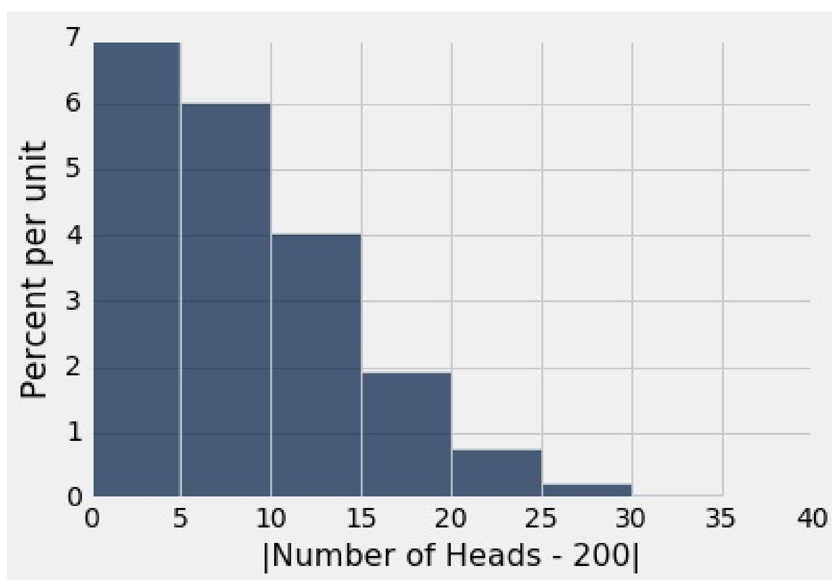
假设你的数据基于 400 个硬币的投掷。 你会预计平等的硬币能够在 400 个次投掷中拥有 200 个正面，所以合理的检验统计量就是使用 $\text{test statistic} = |\text{number of heads} - 200|$。

我们可以在均匀的原假设下模拟统计量。

```
coin = make_array('Heads', 'Tails')
num_tosses = 400

repetitions = 10000
heads = make_array()
for i in np.arange(repetitions):
    tosses = np.random.choice(coin, 400)
    heads = np.append(heads, np.count_nonzero(tosses == 'Heads'))

sampled_stats = abs(heads - 200)
results = Table().with_column('|Number of Heads - 200|', sampled_stats)
results.hist(bins = np.arange(0, 45, 5))
```



如果硬币是不均匀的，那么你预计硬币的数量就不是 200，或者换句话说，如果硬币是均匀的，那么你预计，检验统计量就会大一些。

因此，正如在孟德尔的豌豆花的例子中，P 值是统计量经验分布的右侧尾部的区域。

假设你决定使用 3.5% 的截断值作为 P 值。 那么即使硬币碰巧是均匀的，对于模拟中的 10000 个检验统计量的前 3.5%，你也会得出“不均匀”的结论。

换句话说，如果你用3.5% 的 P 值作为临界值，而硬币恰好是均匀的，那么大概有 3.5% 的概率你会认为硬币是不均匀的。

P 值的截断值是错误概率

上面的例子是一个普遍事实的特例：

如果对 P 值使用 `p%` 的截断值，并且原假设恰好是真的，那么大约有 `p%` 的概率，你的检验就会得出结论：备选假设是正确的。

因此，1% 的截断值比 5% 更保守 - 如果原假设恰好是真的，那么结论为“备选假设”的可能性就会降低。出于这个原因，医学治疗随机对照试验通常使用 1% 作为决定以下两个假设之间的临界值：

原假设：实验没有效果；患者的实验组和对照组的结果之间的观察到的差异，是由于随机性造成的。

备选假设：实验有效果。

这个想法是，控制结论为实验有效，而实际上无效的几率。这减少了给予患者无效治疗的风险。

尽管如此，即使你将截断值设置为 1% 那样低，并且实验没有任何效果，但有大约 1% 的几率得出结论：实验是有效的。这由于机会变异。来自随机样本的数据很可能最终导致你误入歧途。

数据窥探

上面的讨论意味着，如果我们进行 500 个单独的随机对照实验，其中实验实际上没有效果，并且每个实验使用 1% 的截断值，那么通过机会变异，500 个实验中的约 5 个将得出结论：实验确实有效果。

我们可以希望，没有人会对一无所获的实验进行 500 次。但研究人员使用相同的数据测试多个假设并不罕见。例如，在一项关于药物作用的随机对照试验中，研究人员可能会测试该药物是否对各种不同疾病有影响。

现在假设药物对任何东西都没有影响。只是机会变异，一小部分的测试可能会得出结论，它确实有效果。所以，当你阅读一篇使用假设检验的研究，并得出实验有效的结论时，总是询问研究人员，在发现所报告的效果之前，究竟检验了多少种不同的效果。

如果研究人员在找到给出“高度统计学显著”的结论之前，进行了多个不同的检验，请谨慎使用结果。这项研究可能会受到数据窥探的影响，这实际上意味着将数据捏造成一个假象。

在这种情况下，验证报告结果的一种方法是，复制实验并单独检验该特定效果。如果它再次表现为显著，就验证了原来的结论。

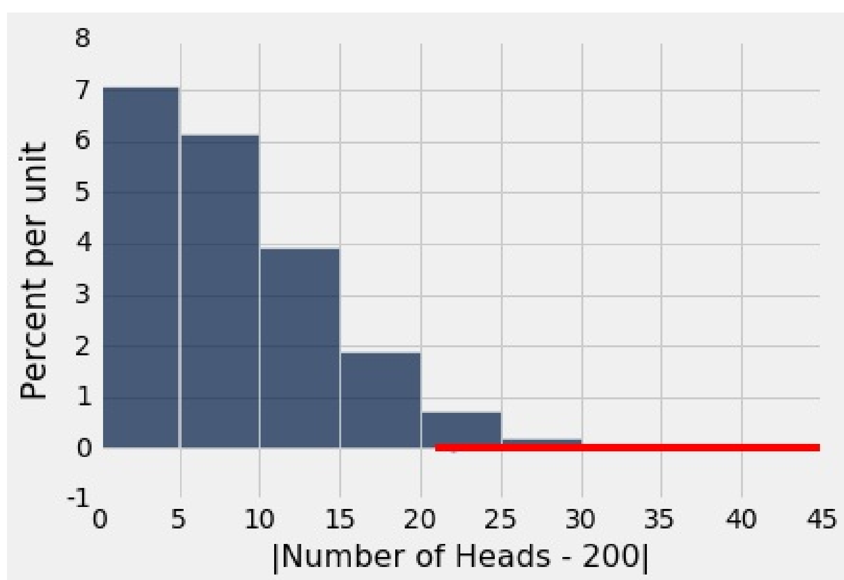
技术注解：其他类型的错误

当然，还有另外一种错误：认为治疗什么也不做，事实上它做了一些事情。近似这个错误超出了本节的范围。要知道，如果你建立你的测试来减少两个错误之一，你几乎总是增加另一个。

技术注解：识别拒绝域

在上面的硬币投掷的例子中，我们基于 400 次投掷，使用 P 值的 3.5 倍的截断值来测试硬币的平等性。检验统计量是 $|\text{number of heads} - 200|$。我们在平等的原假设下模拟了这个统计量。

由于所有统计数据的前 3.5%，检验的结论是硬币是不平等的，在下面展示为红色。



从图中可以看出，在平等的原假设下，大约前 3.5% 的检验统计量的值大于 20。你也可以通过求出这些值的比例来确认：

```
results.where('|Number of Heads - 200|', are.above_or_equal_to(21)).num_rows/results.n
um_rows
0.0372
```

也就是说，如果检验统计量是 21 或更高，那么以 3.5% 的截断点，你会得出结论：硬币是不公平的。

也就是说，如果检验统计量是 21 或更高，你将拒绝原假设。因此，“21 以上”的范围被称为该检验的拒绝域。它对应的正面数量是 221 及以上，或者是 179 及以下。

如果你没有在直方图上将其标记为红色，你将如何找到这些值？百分位数函数在这里派上用场。它需要你尝试查找的百分比水平以及包含数据的数组。统计量的“前 3.5%”对应于统计量的第 96.5 个百分点：

```
percentile(96.5, results.column(0))
21.0
```

注意。由于“重复”（即数据中的几个相同的值）和数据数组的任意长度，百分位数并不总是那么整齐。在本课程的后面，我们将给出一个涵盖所有情况的百分位数的精确定义。就目前而言，只要认为 `percentile` 函数返回一个答案，与你直觉上看做百分点的东西相近即可。

示例：漏风门

2015 年 1 月 18 日，印第安纳波利斯小马队（Indianapolis Colts）和新英格兰爱国者队（New England Patriots）进行了美式橄榄球大会（AFC）冠军赛，来确定哪支球队将晋级超级碗（Super Bowl）。比赛结束后，有人指责爱国者的橄榄球没有按照规定的要求膨胀，并且更软。这可能是一个优势，因为较软的球可能更容易被捕获。

几个星期以来，美国橄榄球界充满了指责，否认，理论和怀疑：在 20 世纪 70 年代水门事件的政治丑闻之后，新闻界标记了“漏风门”这个话题。国家橄榄球联盟（NFL）委托了独立分析小组。在这个例子中，我们将执行我们自己的数据分析。

压强通常以磅/平方英寸（psi）来衡量。 NFL 规则规定了比赛用球必须充气为 12.5psi 到 13.5psi 的压强。每个队都拥有 12 个球。球队有责任保持自己的球的压强，但比赛官方会检查球。在 AFC 比赛开始之前，所有爱国者的球都在 12.5psi 左右。小马队的大部分球在大约 13.0psi。但是，这些赛前数据没有被记录下来。

在第二节，小马队拦截了一个爱国者的球。在边线上，他们测量了球的压强，并确定它低于 12.5psi 的阈值。他们及时通知了官方。

中场休息时，所有的比赛用球都被收集起来检查。两名官方人员 Clete Blakeman 和 Dyrol Prioleau 测量了每个球的压强。这里是数据；压强的单位是磅/平方英寸。被小马队拦截的爱国者的球在这个时候没有被检查。大多数小马队的球也没有 - 官方只是耗完了时间，为了下半场的开始，不得不交出了这些球。

```
football = Table.read_table('football.csv')
football = football.drop('Team')
football.show()
```

Ball	Blakeman	Prioleau
Patriots 1	11.5	11.8
Patriots 2	10.85	11.2
Patriots 3	11.15	11.5
Patriots 4	10.7	11
Patriots 5	11.1	11.45
Patriots 6	11.6	11.95
Patriots 7	11.85	12.3
Patriots 8	11.1	11.55
Patriots 9	10.95	11.35
Patriots 10	10.5	10.9
Patriots 11	10.9	11.35
Colts 1	12.7	12.35
Colts 2	12.75	12.3
Colts 3	12.5	12.95
Colts 4	12.55	12.15

对于被检查的 15 个球中的每一个，两名官员获得了不同的结果。 在同一物体上重复测量得到不同的结果并不少见，特别是当测量由不同的人进行时。 所以我们将每个球赋为这个球上进行的两次测量的平均值。

```
football = football.with_column(
    'Combined', (football.column(1)+football.column(2))/2
    )
football.show()
```

Ball	Blakeman	Prioleau	Combined
Patriots 1	11.5	11.8	11.65
Patriots 2	10.85	11.2	11.025
Patriots 3	11.15	11.5	11.325
Patriots 4	10.7	11	10.85
Patriots 5	11.1	11.45	11.275
Patriots 6	11.6	11.95	11.775
Patriots 7	11.85	12.3	12.075
Patriots 8	11.1	11.55	11.325
Patriots 9	10.95	11.35	11.15
Patriots 10	10.5	10.9	10.7
Patriots 11	10.9	11.35	11.125
Colts 1	12.7	12.35	12.525
Colts 2	12.75	12.3	12.525
Colts 3	12.5	12.95	12.725
Colts 4	12.55	12.15	12.35

一眼望去，爱国者队的压强显然低于小马队。 由于一些放气在比赛过程中是正常的，独立分析师决定计算距离比赛开始的压强下降值。 回想一下，爱国者的球开始时是大约 12.5psi，小马队的球是大约 13.0psi。 因此爱国者球的压强下降值计算为 12.5 减中场时的压强，小马队的球的压强下降值为 13.0 减半场的压强。

我们来构建两张表，一张是爱国者的数据，一张是小马的。 每张表的最后一列是距离开始的压强下降值。

```
patriots = football.where('Ball', are.containing('Patriots'))
patriots = patriots.with_column('Drop', 12.5-patriots.column('Combined'))
patriots.show()
```

Ball	Blakeman	Prioleau	Combined	Drop
Patriots 1	11.5	11.8	11.65	0.85
Patriots 2	10.85	11.2	11.025	1.475
Patriots 3	11.15	11.5	11.325	1.175
Patriots 4	10.7	11	10.85	1.65
Patriots 5	11.1	11.45	11.275	1.225
Patriots 6	11.6	11.95	11.775	0.725
Patriots 7	11.85	12.3	12.075	0.425
Patriots 8	11.1	11.55	11.325	1.175
Patriots 9	10.95	11.35	11.15	1.35
Patriots 10	10.5	10.9	10.7	1.8
Patriots 11	10.9	11.35	11.125	1.375

```
colts = football.where('Ball', are.containing('Colts'))
colts = colts.with_column('Drop', 13.0-colts.column('Combined'))
colts
```

Ball	Blakeman	Prioleau	Combined	Drop
Colts 1	12.7	12.35	12.525	0.475
Colts 2	12.75	12.3	12.525	0.475
Colts 3	12.5	12.95	12.725	0.275
Colts 4	12.55	12.15	12.35	0.65

看起来好像爱国者的漏气比小马队更大。 自然统计量是两个平均漏气之间的差异。 我们将处理它，但你可以自由地用其他自然统计量重复分析，例如整体平均漏气与爱国者之间的差异。

```
patriots_mean = patriots.column('Drop').mean()
colts_mean = colts.column('Drop').mean()

observed_statistic = patriots_mean - colts_mean
observed_statistic
0.733522727272805
```

这种正面的差异反映了这样的事实，即爱国者的球的平均压强下降值大于小马队。

难道这个差异是偶然的，还是爱国者的下降值太大？ 这个问题非常类似于我们之前问过的问题，关于一个大班中的一个小组的成绩。就像我们在这个例子中所做的那样，我们将建立原假设。

原假设：爱国者的下降值就是 15 次下降值中的，大小为 11 的随机样本。 由于机会变异，均值比小马队高。

备选假设：爱国者的下降值太大，并不仅仅是机会变异的结果。

如果原假设是真的，那么爱国者的下降值就可以对比从 15 次下降值随机不带放回抽取的 11 个。 所以让我们创建一个，含有所有 15 个下降值，并从中随机抽取。

```
drops = Table().with_column(
    'Drop', np.append(patriots.column('Drop'), colts.column('Drop'))
)
drops.show()
```

Drop
0.85
1.475
1.175
1.65
1.225
0.725
0.425
1.175
1.35
1.8
1.375
0.475
0.475
0.275
0.65

```
drops.sample(with_replacement=False).show()
```

Drop
1.225
1.175
1.175
0.475
1.375
0.425
0.85
0.65
1.35
1.65
0.725
0.475
1.475
1.8
0.275

注意 `sample` 的使用没有带样本大小。 这是因为 `sample` 使用的默认样本大小是表格的行数；如果你不指定样本大小，则会返回与原始表格大小相同的样本。 这对于我们的目的非常理想，因为当你不放回抽样时（通过指定 `with_replacement = False` ），并且次数与行数相同，最终会对所有行进行随机洗牌。 运行几次该单元格来查看输出如何变化。

我们现在可以使用打乱表的前 11 行作为原假设下的爱国者的下降值的模拟。 剩下的四行形成了对应的小马队的下降值的模拟。 我们可以使用这两个模拟数组来模拟我们在原假设下的检验统计量。

```
shuffled = drops.sample(with_replacement=False)

new_patriots = shuffled.take(np.arange(11))
new_patriots_mean = new_patriots.column(0).mean()

new_colts = shuffled.take(np.arange(11, drops.num_rows))
new_colts_mean = new_colts.column(0).mean()

simulated_stat = new_patriots_mean - new_colts_mean
simulated_stat
-0.7068181818181818212
```

运行几次该单元格来查看检验统计量的变化情况。 请记住，模拟是在原假设下，即爱国者的下降值类似于随机抽样的 15 个下降值。

现在是我们熟悉的步骤了。 我们将在院假设下重复模拟检验统计量。 模拟结束时，数组的 `simulated_statistics` 将包含所有模拟的检验统计量。

```
simulated_statistics = make_array()
repetitions = 10000

for i in np.arange(repetitions):
    shuffled = drops.sample(with_replacement=False)
    new_patriots_mean = shuffled.take(np.arange(11)).column(0).mean()
    new_colts_mean = shuffled.take(np.arange(11, drops.num_rows)).column(0).mean()
    new_statistic = new_patriots_mean - new_colts_mean
    simulated_statistics = np.append(simulated_statistics, new_statistic)
```

现在对于经验 P 值，这是一个几率（在原假设下计算），所得的检验统计量等于观察到统计量，或者更加偏向备选假设方向。 为了弄清楚如何计算它，重要的是要回忆另一个假设：

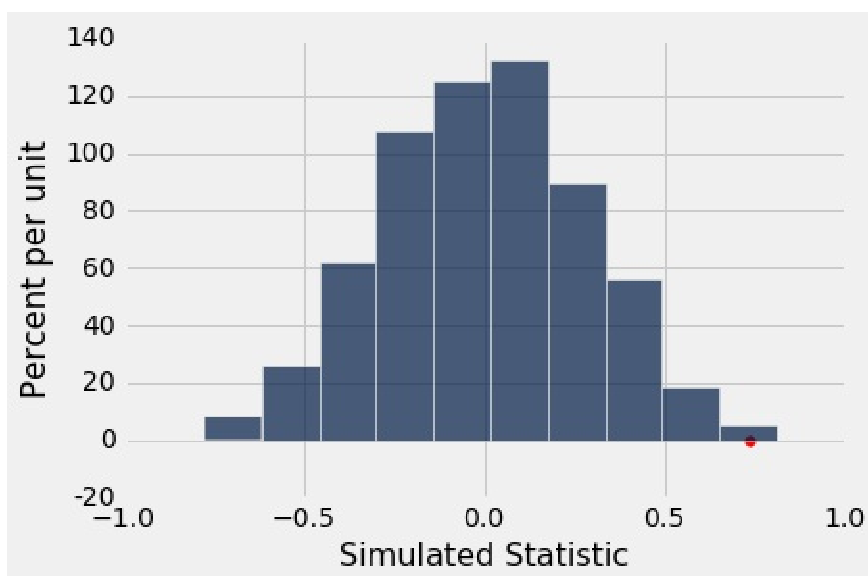
备选假设：爱国者的下降值太大，并不仅仅是机会变异的结果。

“备选假设的方向”是爱国者的下降值很大，对应我们的检验统计量，“爱国者的均值减去小马队的均值”较大。 所以 P 值是几率（在原假设下计算），所得检验统计量大于等于我们 0.7335227272727 2805。

```
empirical_P = np.count_nonzero(simulated_statistics >= observed_statistic)/repetitions
empirical_P
0.0027
```

这是一个非常小的 P 值。 为了观察它，下面是原假设下检验统计量的经验分布，其中观察到的统计量标在横轴上。

```
print('Observed Statistic:', observed_statistic)
print('Empirical P:', empirical_P)
results = Table().with_column('Simulated Statistic', simulated_statistics)
results.hist()
plots.scatter(observed_statistic, 0, color='red', s=30);
Observed Statistic: 0.733522727273
Empirical P: 0.0027
```



请注意，分布大部分集中在 0 左右。在原假设下，爱国者的下降值是所有 15 下降值的随机样本，因此小马对也是如此。 所以这两组下降值的平均值应该大致相等，因此它们的差值应该在 0 左右。

但是检验统计量的观察值离分布的中心还有很远的距离。 使用什么是“小”的任何合理的截断值，经验 P 值都是小的。 所以我们最终拒绝原假设的随机性，并得出结论，爱国者的下降值太大，并不单独反映机会变异。

独立的调查小组以数种不同的方式分析数据，并考虑到物理定律。最后的报告说：

> “爱国者比赛用球的平均压降超过了小马队的球的平均压降 0.45psi 至 1.02psi，这取决于所使用的测量仪的各种可能的假设，并假设爱国者的球的初始压强为 12.5psi，小马队的球是 13.0psi。”
>
> - 2015 年 1 月 18 日，由 NFL 委托对 AFC 冠军赛的调查报告

我们的分析显示，平均压降约为 0.73psi，接近“0.45 至 1.02psi”的中心，因此与官方分析一致。

请记住，我们对假设的检验并没有确定差异不是偶然的原因。 建立因果关系通常比进行假设检验更为复杂。

但足球世界里最重要的问题是因果关系：问题是爱国者足球的压强过大是否是故意的。 如果你对调查人员的答案感到好奇，这里是完整的报告。

十一、估计

> 原文：Estimation
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

在前一章中，我们开始开发推断思维的方法。特别是，我们学会了如何使用数据，在世界的两个假设之间做决策。但是我们通常只想知道，某件事情有多大。

例如，在前面的章节中，我们调查了敌人可能拥有的战机数量。在选举年，我们可能想知道有多少选民赞成特定候选人。为了评估目前的经济状况，我们可能会对美国家庭年收入的中位数感兴趣。

在本章中，我们将开发一种估计未知参数的方法。请记住，参数是总体相关的数值。

要弄清参数的值，我们需要数据。如果我们有整个人口的相关数据，我们可以简单地计算参数。

但是，如果人口非常庞大（例如，如果它由美国的所有家庭组成），那么收集整个人口的数据可能过于昂贵和耗时。在这种情况下，数据科学家依赖从人口中随机抽样。

这导致了一个推断问题：如何根据随机样本中的数据，对未知参数做出正确的结论？我们将用推断思维来回答这个问题。

基于随机样本的统计量可能是总体中未知参数的合理估计。例如，你可能希望使用家庭样本的年收入中位数，来估计美国所有家庭的年收入中位数。

但任何统计量的值都取决于样本，样本基于随机抽取。所以每次数据科学家得到了一个基于随机样本的估计，他们都面临一个问题：

“如果样本是不同的，这个估计有多大的不同呢？”

在本章中，你将学习一种回答这个问题的方法。答案将为你提供工具来估算数值参数，并量化估算中的误差量。

我们将以百分位数开始。最有名的百分位数是中位数，通常用于收入数据的摘要。在我们即将开发的估计方法中，其他百分位数也是非常重要的。所以我们一开始要仔细定义百分位数。

百分位数

数值数据可以按照升序或降序排序。因此，数值数据集的值具有等级顺序。百分位数是特定等级的值。

例如，如果你的考试成绩在第 95 个百分位，一个常见的解释是只有 5% 的成绩高于你的成绩。中位数是第 50 个百分位；通常假定数据集中 50% 的值高于中值。

但是，给予百分位一个精确定义，适用于所有等级和所有列表，需要一些谨慎。为了明白为什么，考虑一个极端的例子，一个班级的所有学生在考试中得分为 75 分。那么 75 是中位数的自然候选，但是 50% 的分数高于 75 并不是真的。另外，75 同样是第 95 个或第 25 个百分位数，或任何其他百分位数的自然候选。在定义百分位数时，必须将重复 - 也就是相同的数据值 - 考虑在内。

当相关的索引不明确时，你还必须小心列表到底有多长。例如，10 个值的集合的第 87 个百分位数是多少？有序集合的第 8 个值，还是第 9 个，还是其中的某个位置？

数值的例子

在给出所有百分位数的一般定义之前，我们将把数值集合的第80个百分点定义为集合中的（一定条件的）最小值，它至少与所有值的 80% 一样大。

例如，考虑非洲，南极洲，亚洲，北美洲和南美洲五大洲的大小，四舍五入到最接近的百万平方英里。

```
sizes = make_array(12, 17, 6, 9, 7)
```

第 80 个百分位数是（一定条件的）最小值，至少和 80% 的值一样大，也就是五个元素的五分之四。等于 12：

```
np.sort(sizes)
array([ 6,  7,  9, 12, 17])
```

第 80 个百分位数是列表中的一个值，也就是 12。你可以看到，80% 的值小于等于它，并且它是列表中满足这个条件的最小值。

与之类似，第 70 个百分位数是该集合中（一定条件的）最小值，至少与 70% 的元素一样大。 现在 5 个元素中的 70% 是“3.5 个元素”，所以第 70 个百分位数是列表中的第 4 个元素。 它是 12，与这些数据的第 80 百分位数相同。

`percentile` 函数

`percentile` 函数接受两个参数：一个 0 到 100 之间的等级，和一个数组。它返回数组相应的百分位数。

```
percentile(70, sizes)
12
```

一般定义

令`p`为 0 到 100 之间的数字。集合的第`p`个百分位数是集合中的（一定条件）的最小值，它至少与`p%`的所有值一样大。

通过这个定义，可以计算任何值的集合的任何 0 到 100 之间的百分位数，并且它始终是集合的一个元素。

实际上，假设集合中有`n`个元素。 要找到第`p`个百分位数：

- 对集合升序排序。
- 计算`n`的`p%`：`(p/100) * n`。叫做`k`。
- 如果`k`是一个整数，则取有序集合的第`k`个元素。
- 如果`k`不是一个整数，则将其四舍五入到下一个整数，并采用有序集合的那个元素。

示例

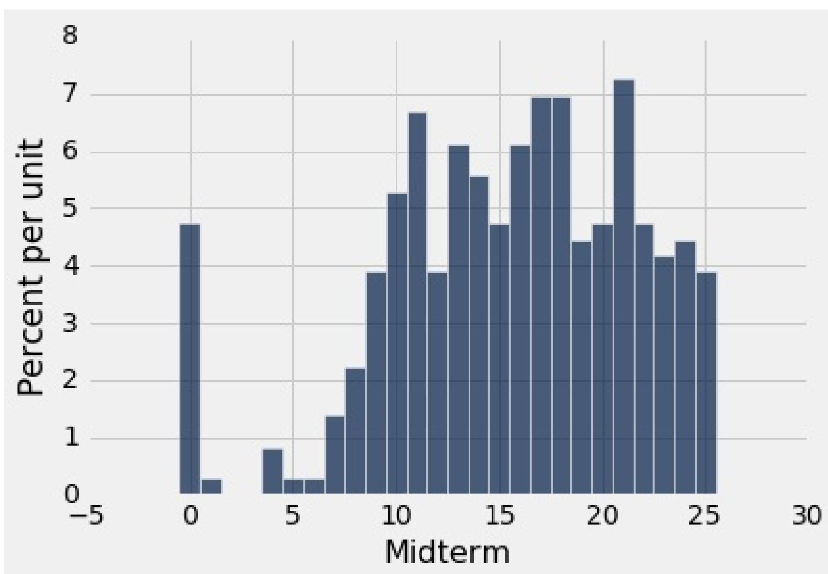
`scores_and_sections`表包含 359 名学生，每个学生一行。 列是学生的讨论分组和期中分数。

```
scores_and_sections = Table.read_table('scores_by_section.csv')
scores_and_sections
```

Section	Midterm
1	22
2	12
2	23
2	14
1	20
3	25
4	19
1	24
5	8
6	14

（省略了 349 列）

```
scores_and_sections.select('Midterm').hist(bins=np.arange(-0.5, 25.6, 1))
```



分数的第 85 个百分位数是多少？ 为了使用 `percentile` 函数，创建包含期中分数的数组 `scores`，并找到第 85 个百分位数：

```
scores = scores_and_sections.column(1)
percentile(85, scores)
22
```

根据 `percentile` 函数，第 85 个百分点数是 22。为了检查这是否符合我们的新定义，我们直接应用定义。

首先，把分数升序排列：

```
sorted_scores = np.sort(scores_and_sections.column(1))
```

数组中有 359 个分数。所以下面，计算 359 的 85%，它是 305.15。

```
0.85 * 359
305.15
```

这不是一个整数。 根据我们的定义，中位数是 `sorted_scores` 的第 306 个元素，按 Python 的索引约定，它是数组的第 305 项。

```
# The 306th element of the sorted array

sorted_scores.item(305)
22
```

它和我们通过使用 `percentile` 得到的答案一样。以后，我们会仅仅使用 `percentile` 。

四分位数

数值集合的第一个四分位数是第 25 个百分分数。 这个术语（quartile）来自第一个季度（quarter）。 第二个四分位数是中位数，第三个四分位数是第 75 个百分位数。

对于我们的分数数据，这些值是：

```
percentile(25, scores)
11
percentile(50, scores)
16
percentile(75, scores)
20
```

分数的分布有时归纳为“中等 50%”区间，在第一和第三个四分位数之间。

自举法

一个数据科学家正在使用随机样本中的数据来估计未知参数。她使用样本来计算用作估计值的统计量。

一旦她计算出了统计量的观察值，她就可以把它作为她的估计值，然后顺其自然。 但她是一名数据科学家。 她知道她的随机样本只是众多可能的随机样本之一，因此她的估计只是众多合理估算之一。

这些估计的变化有多大？ 为了回答这个问题，似乎她需要从总体中抽取另一个样本，并根据新样本计算一个新的估计值。 但是她没有资源来回到总体中，再抽取一个样本。

这个数据科学家看起来好像卡住了。

幸运的是，一个叫做自举法的好主意可以帮助她。 由于从总体中生成新样本是不可行的，自举法通过称为重采样的方法生成新的随机样本：新样本从原始样本中随机抽取。

在本节中，我们将看到自举法的工作方式和原因。 在本章的其余部分，我们将使用自举法进行推理。

旧金山市的雇员薪资

SF OpenData 是一个网站，旧金山市和县在上面公开提供他们的一些数据。 其中一个数据集包含城市雇员的薪资数据。 其中包括市营医院的医疗专业人员，警察，消防员，运输工人，民选官员以及市内所有其他雇员。

2015 日历年的薪资数据见表 `sf2015` 。

```
sf2015 = Table.read_table('san_francisco_2015.csv')
sf2015
```

Year Type	Year	Organization Group Code	Organization Group	Department Code	Department
Calendar	2015	2	Public Works, Transportation & Commerce	WTR	PUC Water Department
Calendar	2015	2	Public Works, Transportation & Commerce	DPW	General Services Agency - Public Works
Calendar	2015	4	Community Health	DPH	Public Health
Calendar	2015	4	Community Health	DPH	Public Health
Calendar	2015	2	Public Works, Transportation & Commerce	MTA	Municipal Transportation Agency
Calendar	2015	1	Public Protection	POL	Police
Calendar	2015	4	Community Health	DPH	Public Health
Calendar	2015	2	Public Works, Transportation & Commerce	MTA	Municipal Transportation Agency
Calendar	2015	6	General Administration & Finance	CAT	City Attorney
Calendar	2015	3	Human Welfare & Neighborhood Development	DSS	Human Services

（省略了 42979 行）

共有 42,979 名员工，每个人一行。有许多列包含市政部门隶属关系的信息，以及员工薪酬方案不同部分的详细信息。这是对应市长 Ed Lee 的一行。

```
sf2015.where('Job', are.equal_to('Mayor'))
```

Year Type	Year	Organization Group Code	Organization Group	Department Code	Department
Calendar	2015	6	General Administration & Finance	MYR	Mayor

我们要研究最后一栏，总薪酬。 这是员工的工资加上市政府对退休和福利计划的贡献。

日历年的财务方案有时难以理解，因为它们取决于雇用日期，员工是否在城市内部换工作等等。 例如，`Total Compensation`列中的最低值看起来有点奇怪。

```
sf2015.sort('Total Compensation')
```

Year Type	Year	Organization Group Code	Organization Group	Department Code	Department
Calendar	2015	1	Public Protection	FIR	Fire Department
Calendar	2015	4	Community Health	DPH	Public Health
Calendar	2015	1	Public Protection	JUV	Juvenile Probation
Calendar	2015	6	General Administration & Finance	CPC	City Planning
Calendar	2015	6	General Administration & Finance	CPC	City Planning
Calendar	2015	2	Public Works, Transportation & Commerce	PUC	PUC Public Utilities Commission
Calendar	2015	1	Public Protection	JUV	Juvenile Probation
Calendar	2015	1	Public Protection	ECD	Department of Emergency Management
Calendar	2015	7	General City Responsibilities	UNA	General Fund Unallocated
Calendar	2015	4	Community Health	DPH	Public Health

（省略了 42979 行）

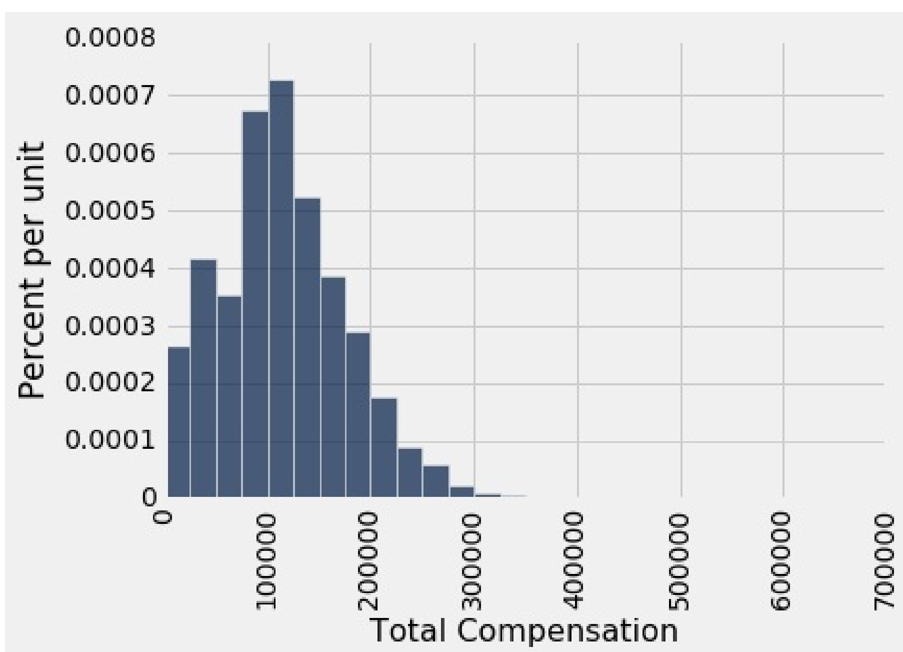
为了便于比较，我们将专注于那些工作时间相当于至少半年的人。 最低工资约为每小时 10 美元，52 周每周 20 小时，工资约为 1 万美元。

```
sf2015 = sf2015.where('Salaries', are.above(10000))
sf2015.num_rows
36569
```

总体和参数

让这张超过 36500 行的表格成为我们的总体。 这是总薪资的直方图。

```
sf_bins = np.arange(0, 700000, 25000)
sf2015.select('Total Compensation').hist(bins=sf_bins)
```



虽然大部分值都低于 300,000 美元，但有一些还是比较高的。 例如，首席投资官的总薪资不多是 65 万美元。 这就是为什么横轴延伸到了 700,000 美元。

```
sf2015.sort('Total Compensation', descending=True).show(2)
```

Year Type	Year	Organization Group Code	Organization Group	Department Code	Department
Calendar	2015	6	General Administration & Finance	RET	Retirement System
Calendar	2015	6	General Administration & Finance	ADM	General Services Agency - City Admin

（省略了 36567 行）

现在让参数为总薪资的中位数。

既然我们有能力从总体中得到所有数据，我们可以简单计算参数：

```
pop_median = percentile(50, sf2015.column('Total Compensation'))
pop_median
110305.78999999999
```

所有员工的薪酬总额的中位数刚刚超过 110,300 美元。

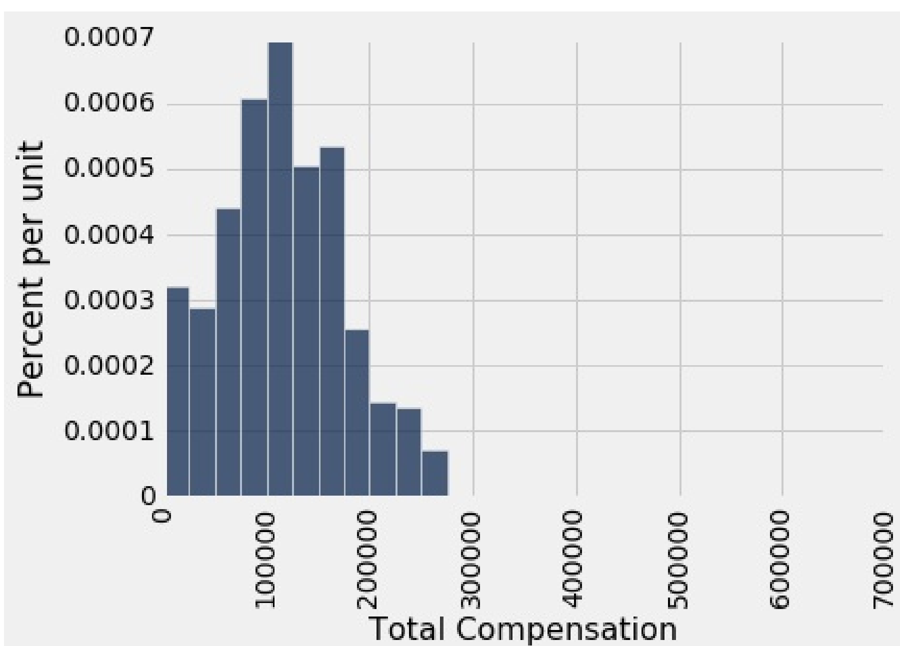
从实际的角度来看，我们没有理由抽取样本来估计这个参数，因为我们只是知道它的值。 但在本节中，我们假装不知道这个值，看看我们如何根据随机样本来估计它。

在后面的章节中，我们将回到现实，在参数未知的情况下工作。 就目前而言，我们是无所不知的。

随机样本和估计

让我们无放回地随机抽取 500 名员工的样本，并将所选员工的总薪酬的中位数作为我们的参数估计量。

```
our_sample = sf2015.sample(500, with_replacement=False)
our_sample.select('Total Compensation').hist(bins=sf_bins)
```




```
est_median = percentile(50, our_sample.column('Total Compensation'))
est_median
113598.99000000001
```

样本量很大。 根据平均定律，样本的分布与总体的分布相似，因此样本中位数与总体中位数相差不大（尽管当然并不完全相同）。

所以现在我们有了参数的估计。 但是，如果样本是不同的，估计的值也会不同。 我们希望能够量化估计的值在不同样本间的差异。 这个变化的测量将有助于我们衡量我们可以将参数估计得多么准确。

为了查看样本有多么不同，我们可以从总体中抽取另一个样本，但这样做就作弊了。 我们正试图模仿现实生活，我们不能掌握所有的人口数据。

用某种方式，我们必须得到另一个随机样本，而不从总体中抽样。

自举法：从样本中重采样

我们所做的是，从样本中随机抽样。 我们知道了，大型随机样本可能类似于用于抽取的总体。 这一观察使得数据科学家可以通过自举来提升自己：抽样过程可以通过从样本中抽样来复制。

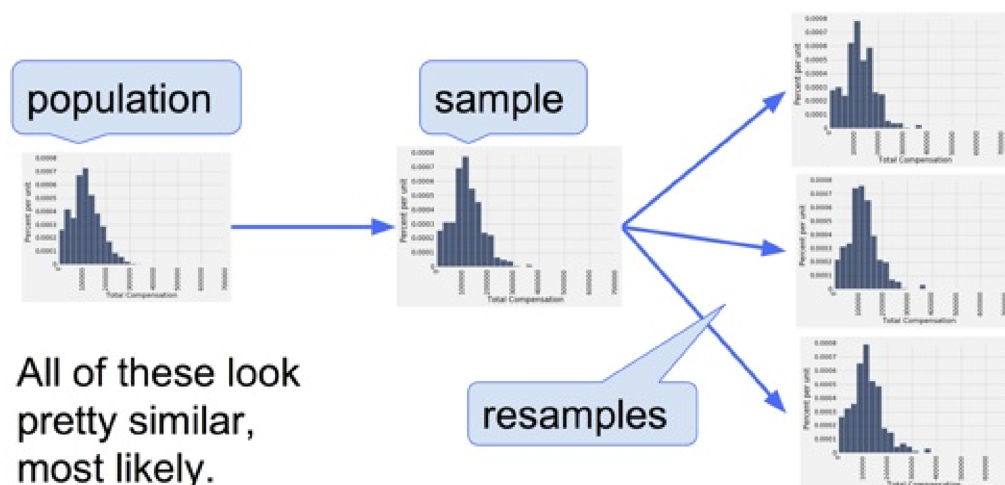
以下是自举法的步骤，用于生成类似总体的另一个随机样本：

- 将原始样本看做总体。
- 从样本中随机抽取样本，与原始样本大小相同。

二次样本的大小与原始样本相同很重要。 原因是估计量的变化取决于样本的大小。 由于我们的原始样本由 500 名员工组成，我们的样本中位数基于 500 个值。 为了看看样本变化多少，我们必须将其与 500 个其他样本的中位数进行比较。

如果我们从大小为 500 的样本中，无放回地随机抽取了 500 次，我们只会得到相同的样本。 通过带放回抽取，我们就可以让新样本与原始样本不同，因为有些员工可能会被抽到一次以上，其他人则完全不会。

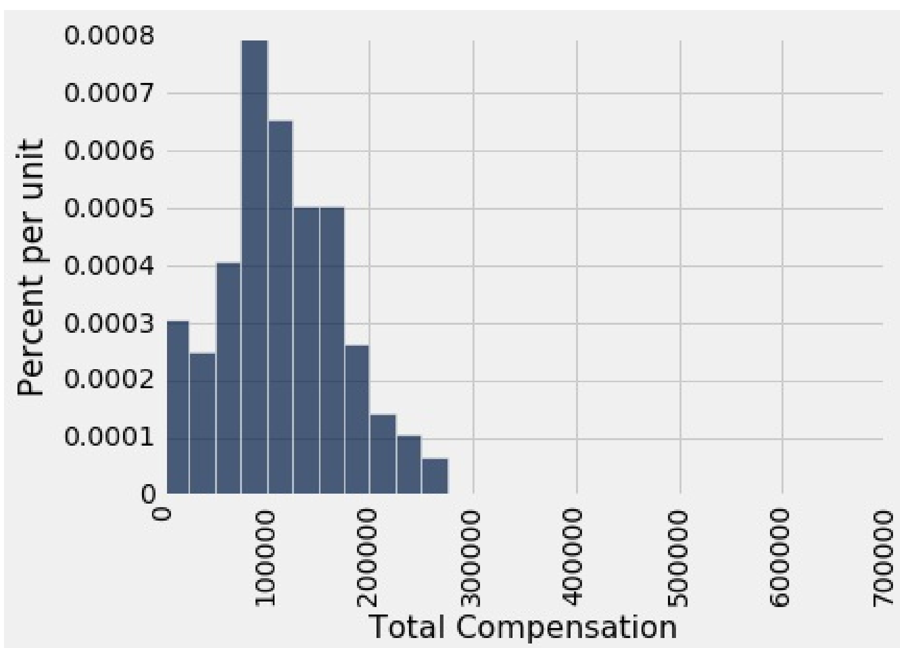
为什么这是一个好主意？ 按照平均定律，原始样本的分布可能与总体相似，所有“二次样本”的分布可能与原始样本相似。 因此，所有二次样本的分布也可能与总体相似。



二次样本的中位数

回想一下，使用 `sample` 方法而没有指定样本大小时，默认情况下样本大小等于用于抽取样本的表的行数。 这是完美的自举！ 这是从原始样本中抽取的一个新样本，以及相应的样本中位数。

```
resample_1 = our_sample.sample()
resample_1.select('Total Compensation').hist(bins=sf_bins)
```




```
resampled_median_1 = percentile(50, resample_1.column('Total Compensation'))
resampled_median_1
110001.16
```

通过重采样，我们有了总体中位数的另一个估计。 通过一次又一次的重采样，我们得到许多这样的估计，因此有了估计的经验分布。

```
resample_2 = our_sample.sample()
resampled_median_2 = percentile(50, resample_2.column('Total Compensation'))
resampled_median_2
110261.39999999999
```

自举样本中位数的经验分布

让我们定义一个函数 `bootstrap_median` ，该函数接受我们的原始样本，包含变量的列的标签，以及我们想要的自举样本的数量，并返回二次样本的相应中值的数组。

每次我们重采样并找到中位数，我们重复自举过程。 所以自举样本的数量将被称为重复数量。

```
def bootstrap_median(original_sample, label, replications):
    """Returns an array of bootstrapped sample medians:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """
    just_one_column = original_sample.select(label)
    medians = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resampled_median = percentile(50, bootstrap_sample.column(0))
        medians = np.append(medians, resampled_median)

    return medians
```

我们现在将自举过程重复 5000 次。 数组 `bstrap_medians` 包含所有 5,000 个自举样本的中位数。 注意代码的运行时间比我们以前的代码要长。 因为要做很多重采样！

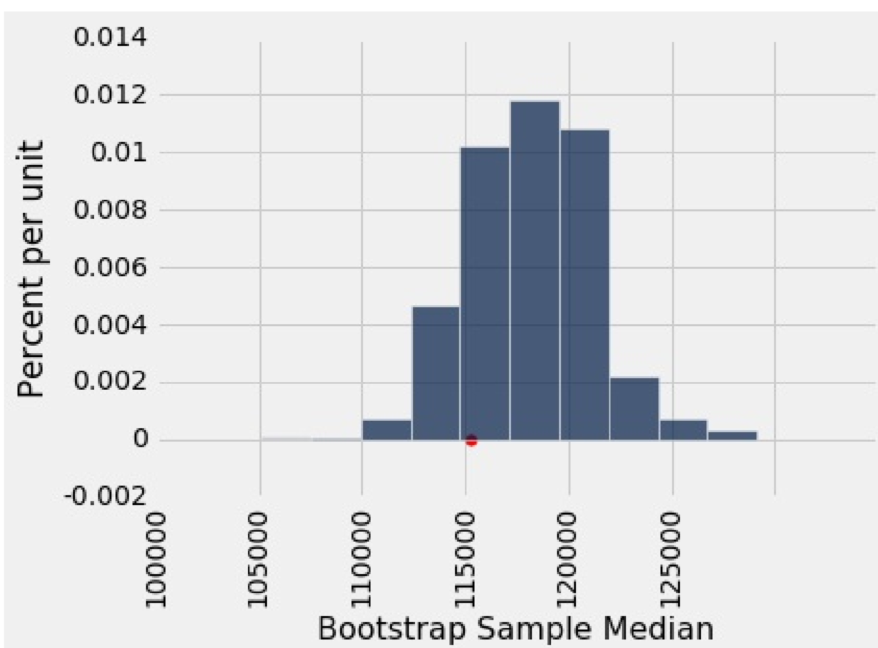
```
bstrap_medians = bootstrap_median(our_sample, 'Total Compensation', 5000)
```

这是 5000 个中位数的直方图。 红点是总体的参数：它是整个总体的中位数，我们碰巧知道但没有在自举过程中使用。

```
resampled_medians = Table().with_column('Bootstrap Sample Median', bstrap_medians)

#median_bins=np.arange(100000, 130000, 2500)
#resampled_medians.hist(bins = median_bins)
resampled_medians.hist()

plots.scatter(pop_median, 0, color='red', s=30);
```



重要的是要记住，红点是固定的：110,305.79 美元，总体的中位数。 经验直方图是随机抽取的结果，将相对于红点随机定位。

请记住，所有这些计算的重点是估计人口中位数，它是红点。我们的估计是所有随机生成的样本中位数，它们的直方图你在上面看到了。我们希望这些估计量包含参数 - 如果没有，它们就脱线了。

估计量是否捕获了参数

红点正好落在二次样本的中位数的经验直方图中间，而不是尾部的几率有多少？要回答这个问题，我们必须定义“中间”。让我们将它看做“红点落在二次样本的中位数的中间 95%”。

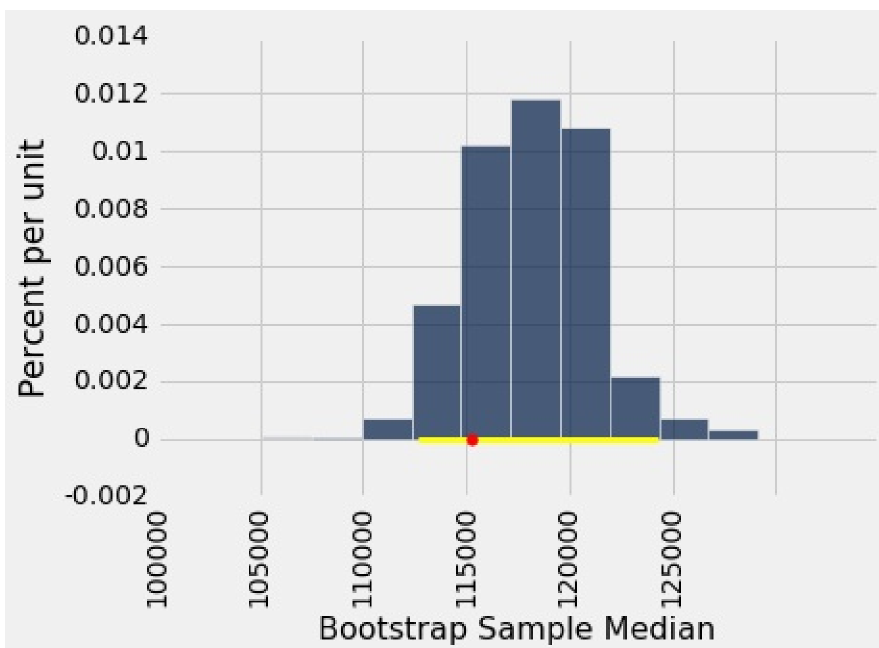
以下是二次采样中位数的“中间 95%”的两端：

```
left = percentile(2.5, bstrap_medians)
left
107652.71000000001
right = percentile(97.5, bstrap_medians)
right
119256.73
```

总体中位数 110,305 美元在这两个数中间。下面的直方图展示了区间和总体中位数。

```
#median_bins=np.arange(100000, 130000, 2500)
#resampled_medians.hist(bins = median_bins)
resampled_medians.hist()

plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=3, zorder=1)
plots.scatter(pop_median, 0, color='red', s=30, zorder=2);
```



我们例子中，估计量的“中间 95%”的区间捕获了参数。但是，这是一个偶然吗？

要查看区间包含参数的频率，我们必须一遍又一遍地运行整个过程。具体而言，我们将重复以下过程 100 次：

- 从总体中抽取一个大小为 500 的原始样本。
- 执行 5000 次重复的自举过程，并生成二次样本的中位数的“中间 95%”的区间。
- 我们最后得到了 100 个区间，并计算其中有多少个包含总体中位数。

剧透警告：自举的统计理论表明，这个数字应该在 95 左右。它可能高于或低于 95，但不会离得太远。

```
# THE BIG SIMULATION: This one takes several minutes.

# Generate 100 intervals, in the table intervals

left_ends = make_array()
right_ends = make_array()

total_comps = sf2015.select('Total Compensation')

for i in np.arange(100):
    first_sample = total_comps.sample(500, with_replacement=False)
    medians = bootstrap_median(first_sample, 'Total Compensation', 5000)
    left_ends = np.append(left_ends, percentile(2.5, medians))
    right_ends = np.append(right_ends, percentile(97.5, medians))

intervals = Table().with_columns(
    'Left', left_ends,
    'Right', right_ends
)
```

对于 100 个重复中的每个，我们得到了一个中位数估计量的区间。

```
intervals
```

Left	Right
100547	115112
98788.4	112129
107981	121218
100965	114796
102596	112056
105386	113909
105225	116918
102844	116712
106584	118054
108451	118421

（省略了 90 行）

良好的区间是那些包含我们试图估计的参数的区间。 通常参数是未知的，但在本节中，我们碰巧知道参数是什么。

```
pop_median
110305.78999999999
```

100 个区间中有多少个包含总体中位数？ 这是左端低于且右端高于总体中位数的区间数量。

```
intervals.where('Left', are.below(pop_median)).where('Right', are.above(pop_median)).n
um_rows
95
```

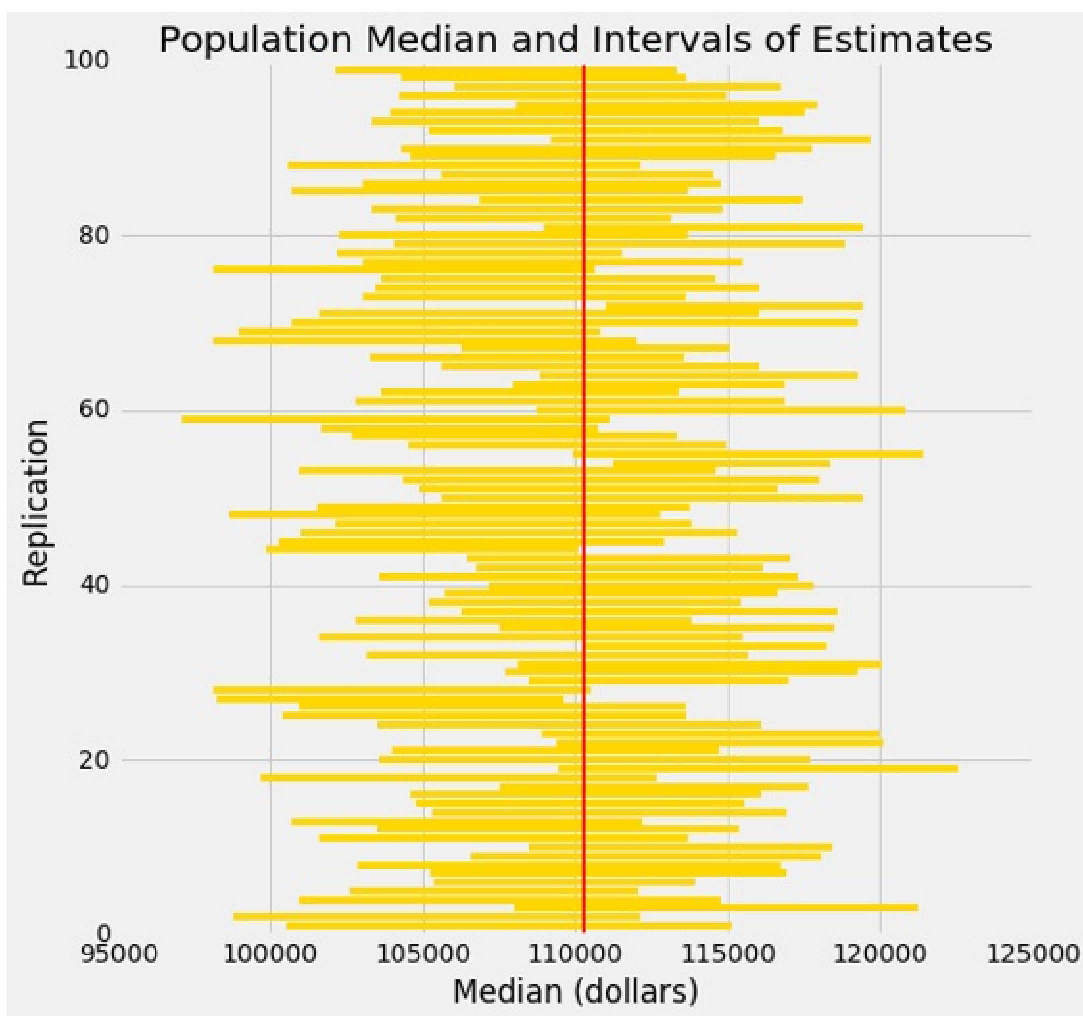
构建所有区间需要花费几分钟时间，但如果你有耐心，请再试一次。最有可能的是，100 个区间中有大约 95 个将是良好的：它们将包含参数。

因为它们有较大的重叠，所以很难在横轴上显示所有的区间 - 毕竟，它们都试图估计相同的参数。下图通过竖直堆叠，在相同轴域上展示的每个区间。纵轴简单地是重复的序号，区间从中生成。

红线是参数所在的位置。良好的区间覆盖了参数；通常有大约 95 个。

如果一个区间不能覆盖这个参数，就是个糟糕的事情。在这个地方，你可以看到红线周围的“亮光”。他们中只有很少 - 通常是大约 5 个 - 但是他们确实存在。

任何基于抽样的方法都有可能脱线。基于随机抽样的方法的优点是，我们可以量化它们可能脱线的频率。



为了总结模拟所示的内容，假设你通过以下过程来估计总体中位数：

从总体中随机抽取一个大样本。 自举你的随机样本，并从新的随机样本中获取估计量。 重复上述步骤数千次，并获得数千个估计量。 挑选所有估计量的“中间 95%”的区间。 这给了你一个估计量的区间。现在，如果重复整个过程 100 次，会得到 100 个区间，那么 100 个区间中的大约 95 个将包含总体的参数。

换句话说，95% 的时间内，这个估计过程捕获了参数。

你可以用一个不同的值代替 95%，只要它不是 100。假设你用 80% 代替了 95%，并保持样本大小为 500。那么你的估计量的区间将比我们这里的模拟要短，因为“中间 80%”是比“中间 95%”更小的范围。只有大约 80% 的区间将包含参数。

置信区间

我们已经开发了一种方法，通过使用随机抽样和自举来估计参数。我们的方法产生一个估计区间，来解释随机样本的机会变异。通过提供一个估计区间而不是一个估计量，我们给自己一些回旋的余地。

在前面的例子中，我们看到我们的估计过程在 95% 的时间内产生了一个良好的区间，一个“良好”的区间就是包含这个参数的区间。对于这个过程的结果很好，我们说我们有 95% 的置信度（信心）。我们的估计区间称为参数的 95% 置信区间，95% 称为区间的置信度。

前一个例子中的情况有点不寻常。因为我们碰巧知道参数的值，所以我们能够检查一个区间是好还是不好，这反过来又帮助我们看到，我们的估计过程每 100 次中有 95 次捕获了参数。

但通常情况下，数据科学家不知道参数的值。这就是他们首先想要估计的原因。在这种情况下，他们通过使用一些方法，类似我们开发的方法，获得未知参数的估计区间。由于统计理论，和我们所看到的演示，数据科学家可以确信，他们产生区间的过程，会以已知百分比的几率，产生一个良好的区间。

总体中位数的置信区间：自举百分位数方法

现在我们使用自举法来估计未知总体的中位数。 数据来自大型医院系统中的新生儿样本; 我们将把它看作是一个简单的随机样本，虽然抽样分多个阶段完成。 Deborah Nolan 和 Terry Speed 的 Stat Labs 拥有一个大数据集的详细信息，这个样本是从中抽取的。

`baby` 表中包含以下母婴偶对的数量：婴儿的出生体重（盎司），孕期天数，母亲的年龄，母亲身高（英寸），孕期体重（磅），母亲是否在孕期吸烟。

```
baby = Table.read_table('baby.csv')
baby
```

Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Maternal Smoker
120	284	27	62	100	False
113	282	33	64	135	False
128	279	28	64	115	True
108	282	23	67	125	True
136	286	25	62	93	False
138	244	33	62	178	False
132	245	23	65	140	False
120	289	25	62	125	False
143	299	30	66	136	True
140	351	27	68	120	False

（省略了 1164 行）

出生体重是新生儿健康的一个重要因素 - 较小的婴儿比较大的婴儿在初期需要更多的医疗护理。 因此，在婴儿出生前估计出生体重是有帮助的。 一种方法是检查出生体重和怀孕天数之间的关系。

这种关系的一个简单的衡量标准是出生体重与怀孕天数的比值。`ratios` 表包含 `baby` 的前两列，以及一列 `ratios`。 这一列的第一个条目按以下方式计算：

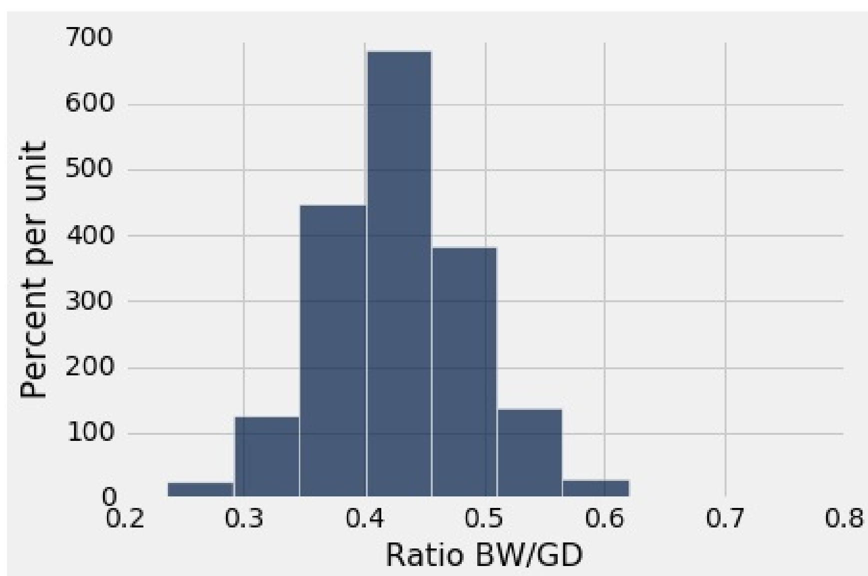
$$\frac{120\text{ ounces}}{284\text{ days}} \approx 0.4225\text{ ounces per day}$$

```
ratios = baby.select('Birth Weight', 'Gestational Days').with_column(
    'Ratio BW/GD', baby.column('Birth Weight')/baby.column('Gestational Days')
)
ratios
```

Birth Weight	Gestational Days	Ratio BW/GD
120	284	0.422535
113	282	0.400709
128	279	0.458781
108	282	0.382979
136	286	0.475524
138	244	0.565574
132	245	0.538776
120	289	0.415225
143	299	0.478261
140	351	0.39886

（省略了 1164 行）

```
ratios.select('Ratio BW/GD').hist()
```



一眼望去，直方图看起来相当对称，密度在 4opd 到 4.5opd 的区间内是最大的。但仔细一看，就可以看出一些比例相当大。比率的最大值刚好超过 0.78opd，几乎是通常值的两倍。

```
ratios.sort('Ratio BW/GD', descending=True).take(0)
```

Birth Weight	Gestational Days	Ratio BW/GD
116	148	0.783784

中位数提供了通常比例的感觉，因为它不受非常大或非常小的比例的影响。样本（比值）的中位数约为 0.429opd。

```
np.median(ratios.column(2))
0.42907801418439717
```

但是总体中位数是多少？我们不知道，所以我们会估计它。

我们的方法将与前一节完全相同。我们将自举样本 5000 次，结果是 5000 个中位数的估计量。我们 95% 的置信区间将是我们所有估计量的“中间 95%”。

回忆前一节定义的 `bootstrap_median` 函数。我们将调用这个函数，并构造总体（比值）中位数的 95% 置信区间。请记住，`ratios` 表包含来自我们的原始样本的相关数据。

```
def bootstrap_median(original_sample, label, replications):

    """Returns an array of bootstrapped sample medians:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    medians = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resampled_median = percentile(50, bootstrap_sample.column(0))
        medians = np.append(medians, resampled_median)

    return medians
# Generate the medians from 5000 bootstrap samples
bstrap_medians = bootstrap_median(ratios, 'Ratio BW/GD', 5000)
# Get the endpoints of the 95% confidence interval
left = percentile(2.5, bstrap_medians)
right = percentile(97.5, bstrap_medians)

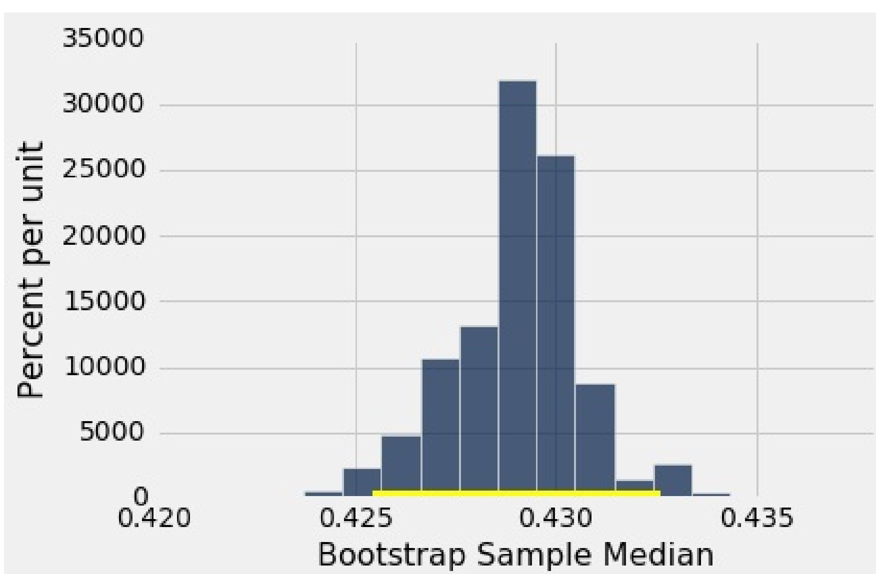
make_array(left, right)
array([ 0.42545455,  0.43262411])
```

95% 置信区间是 0.425opd 到 0.433opd。 我们估计的总体（出生重量与怀孕天数的比值）中位数，在 0.425opd 到 0.433opd 的范围内。

基于原始样本的估计量 0.429 恰好在区间两端的中间，尽管这通常不是真的。

为了使我们的结果可视化，让我们画出我们自举的中位数的经验直方图，并将置信区间置于横轴上。

```
resampled_medians = Table().with_column(
    'Bootstrap Sample Median', bstrap_medians
)
resampled_medians.hist(bins=15)
plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=8);
```



这个直方图和区间就像我们在前一节中绘制的直方图和区间，只有一个很大的区别 - 没有显示参数的红点。 我们不知道这个点应该在哪里，或者它是否在区间中。

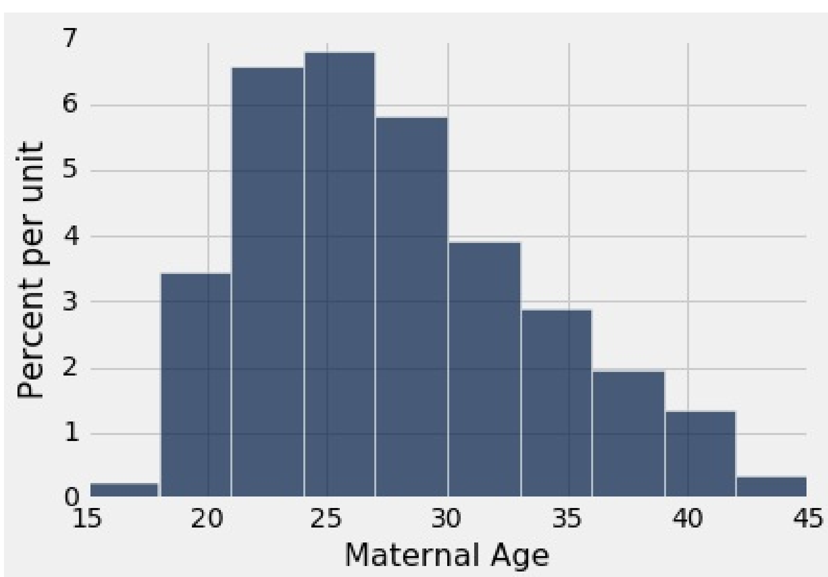
我们只是有一个估计区间。 这是估计量的 95% 置信区间，因为生成它的过程在 95% 的时间中产生了良好的区间。 那肯定是在随机猜测！

请记住，这个区间是一个大约 95% 的置信区间。 计算中涉及到很多近似值。 近似值并不差，但并不准确。

总体均值的置信区间：自举百分位数方法

我们为中位数所做的事情也可以用于均值。 假设我们想估计总体中的母亲的平均年龄。 自然估计量是样本中的母亲的平均年龄。 这是他们的年龄分布，他们的平均年龄大约是 27.2 岁。

```
baby.select('Maternal Age').hist()
```




```
np.mean(baby.column('Maternal Age'))
27.228279386712096
```

母亲的平均年龄是多少？ 我们不知道这个参数的值。

我们用自举法来估计未知参数。 为此，我们将编辑 `bootstrap_median` 的代码，而不是定义函数 `bootstrap_mean` 。 代码是相同的，除了统计量是代替中位数的均值，并且收集在一个名为 `means` 而不是 `medians` 的数组中。

```
def bootstrap_mean(original_sample, label, replications):

    """Returns an array of bootstrapped sample means:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    means = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resampled_mean = np.mean(bootstrap_sample.column(0))
        means = np.append(means, resampled_mean)

    return means
# Generate the means from 5000 bootstrap samples
bstrap_means = bootstrap_mean(baby, 'Maternal Age', 5000)

# Get the endpoints of the 95% confidence interval
left = percentile(2.5, bstrap_means)
right = percentile(97.5, bstrap_means)

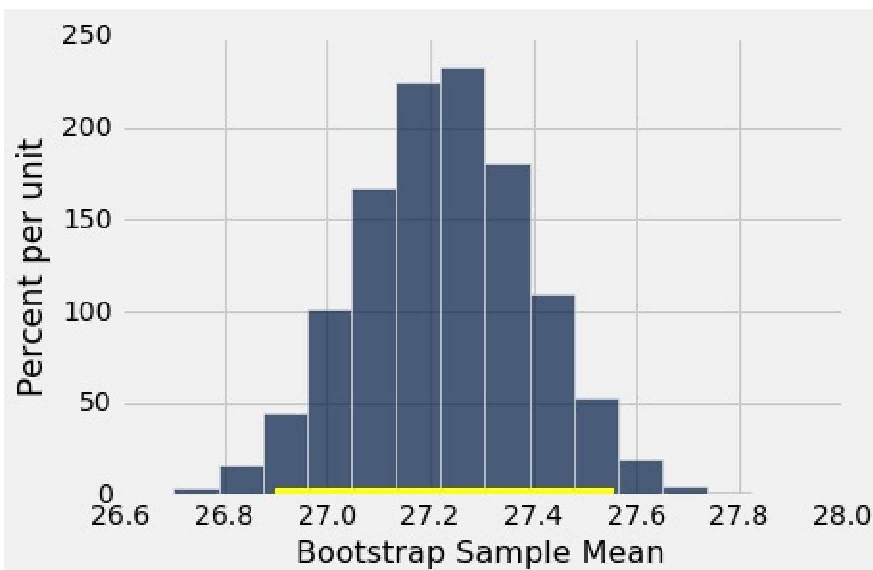
make_array(left, right)
array([ 26.89778535,  27.55962521])
```

95% 置信区间是约 26.9 岁到约 27.6 岁。 也就是说，我们估计的母亲的平均年龄在 26.9 岁到 27.6 岁之间。

注意两端距原始样本均值 27.2 岁的距离。 样本量非常大 - 1174 个母亲 - 所以样本均值变化不大。 我们将在下一章进一步探讨这个观察。

下面显示了 5000 个自举均值的经验直方图，以及总体均值的 95% 置信区间。

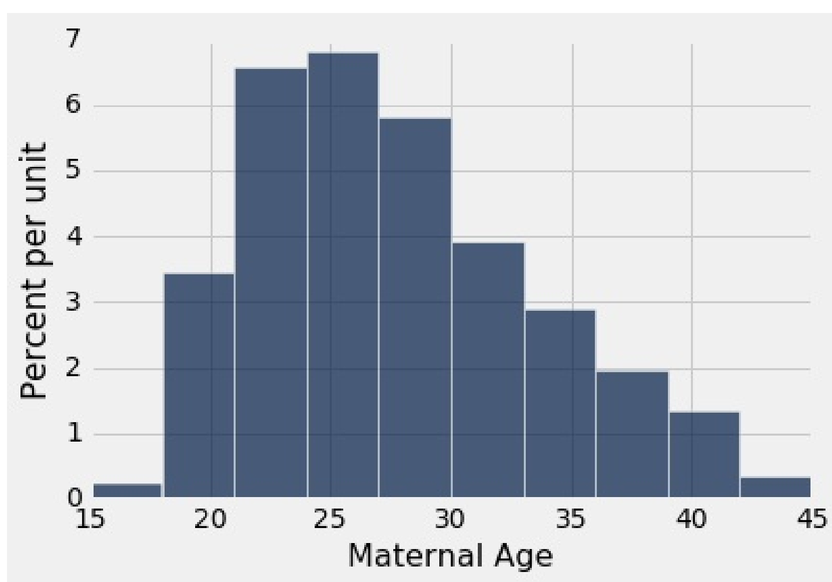
```
resampled_means = Table().with_column(
    'Bootstrap Sample Mean', bstrap_means
)
resampled_means.hist(bins=15)
plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=8);
```



原始样本的均值（27.23 岁）同样接近区间中心。 这并不奇怪，因为每个自举样本都是从相同的原始样本中抽取的。 自举样本的均值大约对称分布原始样本（从其中抽取）的均值的两侧。

还要注意，即使所采样的年龄的直方图完全不是对称的，二次样本的均值的经验直方图也是大致对称的钟形：

```
baby.select('Maternal Age').hist()
```

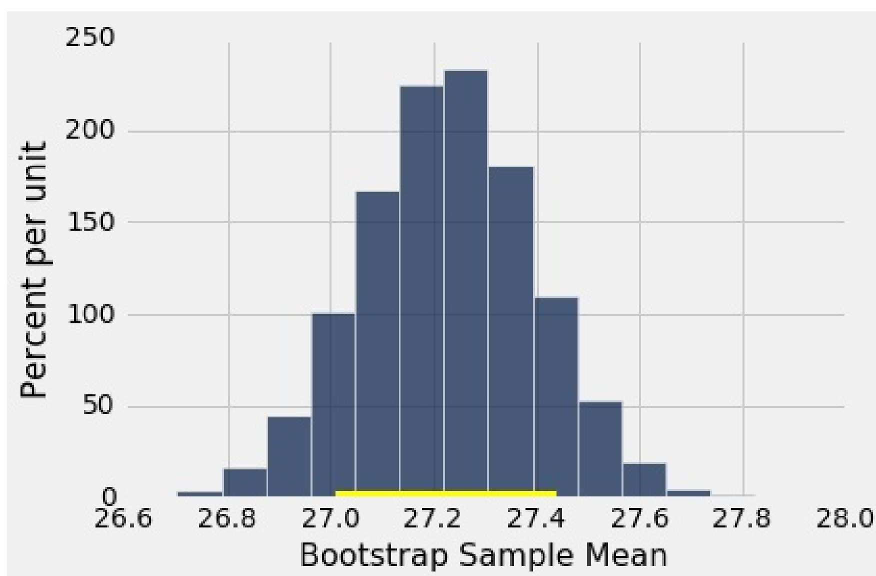


这是概率统计的中心极限定理的结果。 在后面的章节中，我们将看到这个定理是什么。

80% 置信区间

你可以使用自举法来构建任意水平的置信区间。 例如，要为总体中的平均年龄构建 80% 置信区间，可以选取二次样本的均值的“中间 80%”。 所以你会希望为两个尾巴的每一个分配 10%，因此端点是二次样本的均值的第 10 和第 90 个百分位数。

```
left_80 = percentile(10, bstrap_means)
right_80 = percentile(90, bstrap_means)
make_array(left_80, right_80)
array([ 27.01192504,  27.439523  ])
resampled_means.hist(bins=15)
plots.plot(make_array(left_80, right_80), make_array(0, 0), color='yellow', lw=8);
```



这个 80% 置信区间比 95% 置信区间要短得多。 它只是约定 27.0 岁到约 27.4 岁。 虽然这是估计量的较窄区间，你知道这个过程在 80% 的时间内产生良好的区间。

之前过程产生了较宽的区间，但是我们对产生它的过程拥有更高的置信度。

为了以较高的置信度获得较窄的置信区间，你必须从较大的样本开始。 我们将在下一章看到为什么。

总体比例的置信区间：自举百分位数方法

在样本中，39% 的母亲在怀孕期间吸烟。

```
baby.where('Maternal Smoker', are.equal_to(True)).num_rows/baby.num_rows
0.3909710391822828
```

以下对观察很实用，这个比例也可以通过数组操作来计算：

```
smoking = baby.column('Maternal Smoker')
np.count_nonzero(smoking)/len(smoking)
0.3909710391822828
```

> 译者注：
>
> `np.count_nonzero(arr)` 等价于 `np.sum(arr != 0)` 。

总体中有百分之多少的母亲在怀孕期间吸烟？ 这是一个未知的参数，我们可以通过自举置信区间来估计。 这个过程中的步骤与我们用来估计总体均值和中位数的步骤相似。

我们将首先定义一个函数 `bootstrap_proportion` ，返回一个自举样本的比例的数组。 我们再一次通过编辑 `bootstrap_median` 的定义来实现它。 计算中唯一的变化是用二次样本的吸烟者比例代替中位数。 该代码假定数据列由布尔值组成。 其他的改变只是数组的名字，来帮助我们阅读和理解我们的代码。

```
def bootstrap_proportion(original_sample, label, replications):

    """Returns an array of bootstrapped sample proportions:
    original_sample: table containing the original sample
    label: label of column containing the Boolean variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    proportions = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resample_array = bootstrap_sample.column(0)
        resampled_proportion = np.count_nonzero(resample_array)/len(resample_array)
        proportions = np.append(proportions, resampled_proportion)

    return proportions
```

让我们使用 `bootstrap_proportion` 来构建总体的母亲吸烟者百分比的 95% 置信区间。 该代码类似于均值和中位数的相应代码。

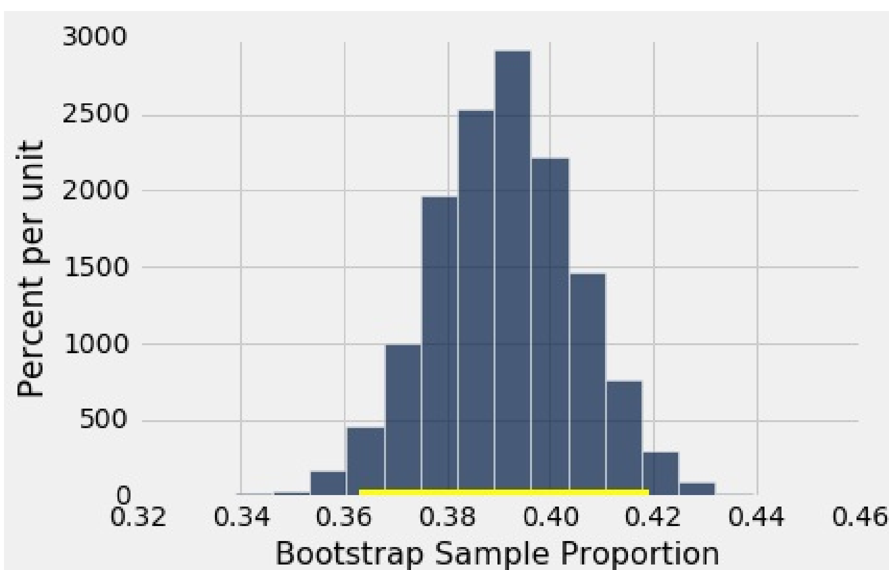
```
# Generate the proportions from 5000 bootstrap samples
bstrap_props = bootstrap_proportion(baby, 'Maternal Smoker', 5000)

# Get the endpoints of the 95% confidence interval
left = percentile(2.5, bstrap_props)
right = percentile(97.5, bstrap_props)

make_array(left, right)
array([ 0.36286201,  0.41908007])
```

置信区间是 36% 到 42%。原始样本的百分比 39% 非常接近于区间的中心。你可以在下面看到：

```
resampled_proportions = Table().with_column(
    'Bootstrap Sample Proportion', bstrap_props
)
resampled_proportions.hist(bins=15)
plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=8);
```



自举法的注意事项

自举法是一个优雅而强大的方法。在使用之前，记住一些要点非常重要。

以大型随机样本开始。如果你不这样做，该方法可能无法正常工作。它的成功基于大型随机样本（因此也从样本中重采样）。平均定律说，如果随机样本很大，这很可能是眞的。

为了近似统计量的概率分布，最好多次复制重采样过程。数千次重复将产生样本中位数分布的正确近似，特别是如果总体分布存在峰值并且不是非常不对称的话。在我们的例子中，我们使用了 5000 次重复，但一般会推荐 10000 次。

自举百分位数方法适用于基于大型随机样本，估计总体中位数或均值。但是，它也有其局限性，所有的估计方法也是如此。例如，在以下情况下，它预期没有效果。

- 目标是估计总体中的最小值或最大值，或非常低或非常高的百分位数，或受总体中稀有元素影响较大的参数。
- 统计量的概率分布不是近似钟形的。
- 原始样本非常小，比如 10 或 15。

使用置信区间

```
def bootstrap_median(original_sample, label, replications):

    """Returns an array of bootstrapped sample medians:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    medians = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resampled_median = percentile(50, bootstrap_sample.column(0))
        medians = np.append(medians, resampled_median)

    return medians
def bootstrap_mean(original_sample, label, replications):

    """Returns an array of bootstrapped sample means:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    means = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resampled_mean = np.mean(bootstrap_sample.column(0))
        means = np.append(means, resampled_mean)

    return means
def bootstrap_proportion(original_sample, label, replications):

    """Returns an array of bootstrapped sample proportions:
    original_sample: table containing the original sample
    label: label of column containing the Boolean variable
    replications: number of bootstrap samples
    """

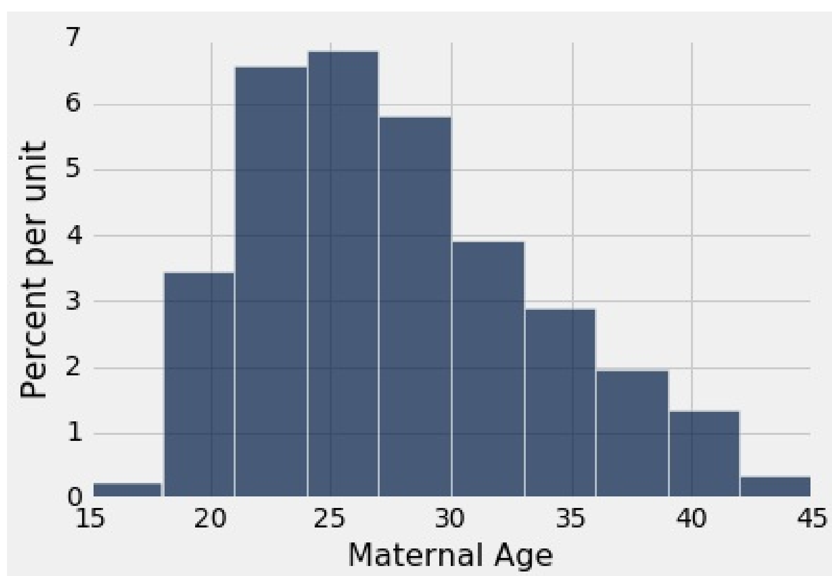
    just_one_column = original_sample.select(label)
    proportions = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resample_array = bootstrap_sample.column(0)
        resampled_proportion = np.count_nonzero(resample_array)/len(resample_array)
        proportions = np.append(proportions, resampled_proportion)

    return proportions
```

置信区间只有一个目的 - 根据随机样本中的数据估计未知参数。在最后一节中，我们说区间 `(36%, 42%)` 是总体中吸烟者百分比的约 95% 的置信区间。正式的表述方式为，据我们估计，总体中的吸烟者比例在 36% 到 42% 之间，我们的估计过程在 95% 的时间内是正确的。

克制住将置信区间用于其他目的的冲动，这很重要。例如，回想一下，我们计算了区间 `(26.9 yr, 27.6 yr)`，作为母亲平均年龄的约 95% 的置信区间。区间的一个令人惊讶的常见误用是得出这样的结论，约 95% 的女性在 26.9 岁至 27.6 岁之间。你不需要怎么了解置信区间，来查看这是不是正确的 - 你不会预计 95% 的母亲的年龄在这个较小的范围内。实际上，抽样年龄的直方图显示出相当多的变化。

```
baby = Table.read_table('baby.csv')
baby.select('Maternal Age').hist()
```



抽样年龄的一小部分在 `(26.9, 27.6)` 的区间内，你可能会预计总体中的百分比很小。 区间只是估计一个数字：总体中所有年龄的平均值。

但是，除了仅仅告诉我们这个参数有多大之外，用置信区间来估计一个参数确实有重要的用处。

使用置信区间来检验假设

我们总体（年龄）均值的 95% 置信区间是 26.9 岁到 27.6 岁。假设有人想要测试以下假设：

原假设。人口的平均年龄是 30 岁。

备选假设。人口的平均年龄不是 30 岁。

那么，如果你使用 5% 的截断值作为 P 值，则会拒绝原假设。这是因为总体平均值 30 不在 95% 置信区间内。在 5% 的显着性水平上，30 对于人口平均值并不合理。

置信区间的使用是置信区间和检验之间二元性结果：如果你正在测试总体平均值是否是特定值 x，并且你使用的 5% 截断值作为 P 值，那么如果 x 不在平均值的 95% 置信区间内，你将拒绝原零假设。

这可以由统计理论来确定。在实践中，它只是归结为，检查原假设中指定的值是否在置信区间内。

如果你使用 1% 的截断值作为 P 值，你必须检查，原假设中指定的值是否在总体均值的 99% 置信区间内。

粗略地说，如果样本量很大，这些陈述也适用于总体比例。

虽然我们现在有一种方法，使用置信区间来检验一种特定假设，但是你可能想知道，测试总体（年龄）的均值是否等于 30 的意义。实际上，这个意义并不清楚。但是在某些情况下，对这种假设的检验既自然又有用。

我们将在数据的背景下来研究它，这些数据是霍奇金病治疗的随机对照试验中收集的信息的子集。霍奇金病是一种通常影响年轻人的癌症。这种疾病是可以治愈的，但治疗可能非常艰难。该试验的目的是找出治疗癌症的剂量，并且将对患者的不利影响最小化。

这张表格包含治疗对 22 名患者肺部的影响的数据。这些列是：

- 身高（厘米）
- 覆盖物辐射的测量（颈部，胸部，手臂下）
- 化疗的测量
- 基线下，即在治疗开始时的肺健康得分；较高的分数对应于更健康的肺
- 治疗后 15 个月，相同的肺的健康得分

```
hodgkins = Table.read_table('hodgkins.csv')
hodgkins
```

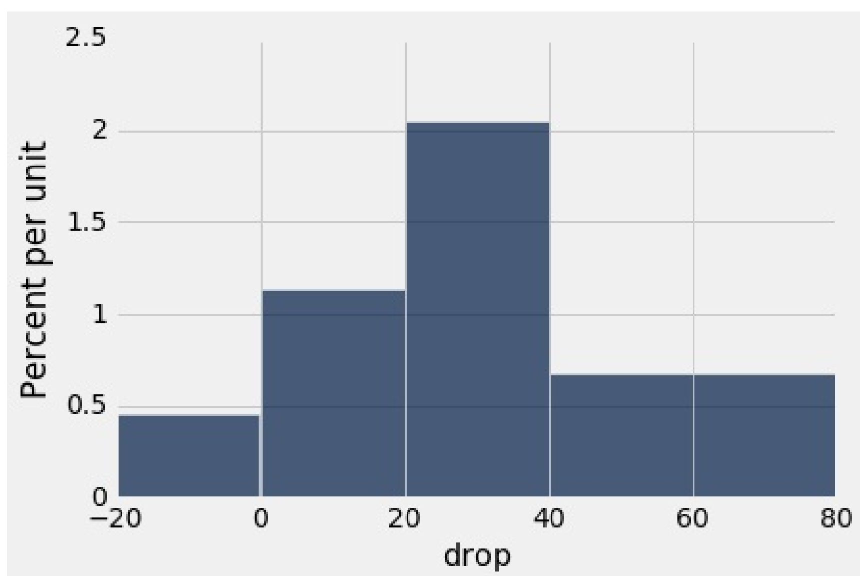
height	rad	chemo	base	month15
164	679	180	160.57	87.77
168	311	180	98.24	67.62
173	388	239	129.04	133.33
157	370	168	85.41	81.28
160	468	151	67.94	79.26
170	341	96	150.51	80.97
163	453	134	129.88	69.24
175	529	264	87.45	56.48
185	392	240	149.84	106.99
178	479	216	92.24	73.43

（省略了 12 行）

我们将比较基准和 15 个月的得分。 由于每行对应一个病人，我们说基线得分的样本和 15 个月得分的样本是成对的 - 它们不是每组 22 个值的两组，而是 22 对值，每个病人一个。

一眼望去，你可以看到，15 个月的得分往往低于基线得分 - 抽样患者的肺似乎在治疗后 15 个月更差。 这个由 `drop` 列主要是正值来证实，它是基线得分减去 15 个月的得分。

```
hodgkins = hodgkins.with_column(
    'drop', hodgkins.column('base') - hodgkins.column('month15')
)
hodgkins
```



height	rad	chemo	base	month15	drop
164	679	180	160.57	87.77	72.8
168	311	180	98.24	67.62	30.62
173	388	239	129.04	133.33	-4.29
157	370	168	85.41	81.28	4.13
160	468	151	67.94	79.26	-11.32
170	341	96	150.51	80.97	69.54
163	453	134	129.88	69.24	60.64
175	529	264	87.45	56.48	30.97
185	392	240	149.84	106.99	42.85
178	479	216	92.24	73.43	18.81

（省略了 12 行）

```
hodgkins.select('drop').hist(bins=np.arange(-20, 81, 20))
```

```
np.mean(hodgkins.column('drop'))
28.615909090909096
```

但是，这可能是机会变异的结果吗？似乎并不如此，但数据来自随机样本。难道在整个人群中，平均下降值只有 0 吗？

为了回答这个问题，我们可以设定两个假设：

原假设：总体（下降值）均值为 0。

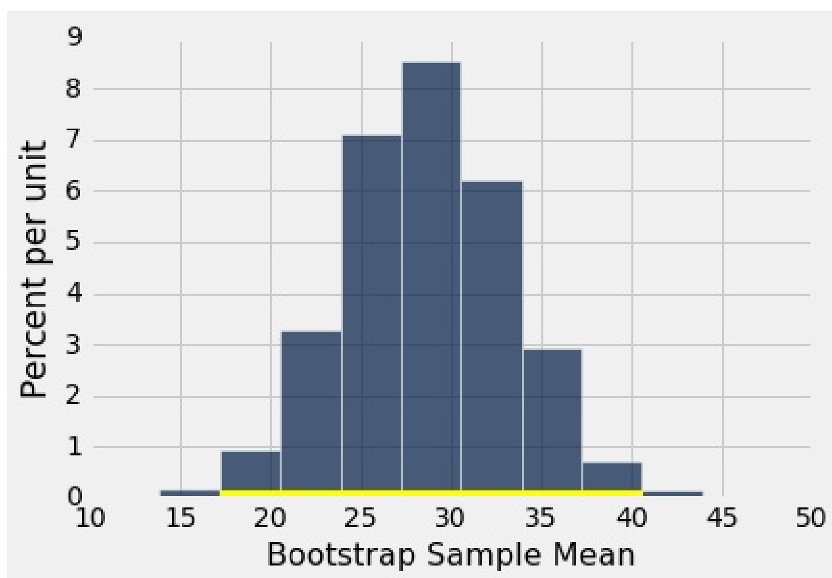
备选假设：总体（下降值）均值不为 0。

为了使用 1% 的截断值作为 P 值来验证这个假设，让我们为总体（下降值）均值构建近似 99% 置信区间。

```
bstrap_means = bootstrap_mean(hodgkins, 'drop', 10000)

left = percentile(0.5, bstrap_means)
right = percentile(99.5, bstrap_means)

make_array(left, right)
array([ 17.25045455,  40.60136364])
resampled_means = Table().with_column(
    'Bootstrap Sample Mean', bstrap_means
)
resampled_means.hist()
plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=8);
```



总体均值的 99% 置信区间是约 17 到约 40。区间不包含 0。因此，我们拒绝原假设。

但是请注意，我们所做的不仅仅是简单得出结论：总体均值不是 0，我们估计了均值的幅度是多大。这比仅仅说“不是 0”更有用。

对于准确性的注解：我们的置信区间相当宽泛，主要有两个原因：

- 置信水平很高（99%）。
- 与我们之前的例子相比，样本量相对较小。

在下一章中，我们将研究样本大小如何影响准确性。我们还将研究，样本均值的经验分布为何经常出现钟形，尽管底层数据的分布根本不是钟形的。

尾注

一个领域的术语通常来自该领域的主要研究人员。首先提出自举技术的 Brad Efron 用了一个美国血统的术语。中国统计学家不甘示弱，提出了自己的方法。

十二、为什么均值重要

> 原文：Why the Mean Matters
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

在这个课程中，我们已经研究了几个不同的统计量，包括总编译距离，最大值，中位数和平均值。在关于随机性的明确假设下，我们绘制了所有这些统计量的经验分布。有些统计量，比如最大和总变异距离，分布明显偏向一个方向。但是，无论研究对象如何，样本均值的经验分布几乎总是接近钟形。

如果随机样本的性质是真的，不管总体如何，它都能成为一个有力的推理工具，因为我们通常不清楚总体中的数据。大型随机样本的均值分布属于这类性质。这就是随机抽样方法广泛用于数据科学的原因。

在本章中，我们将研究均值，以及我们可以说的一些东西，仅仅使用最基本的底层总体的假设。我们要解决的问题包括：

- 均值正好测量了什么？
- 大部分数据与平均值有多接近？
- 样本量如何与样本的均值相关？
- 为什么随机样本的经验分布出现钟形？
- 我们如何有效地使用抽样方法进行推理？

均值的性质

在这个课程中，我们可以互换地使用“average”和“mean”两个单词（译者注，在中文中都译为“均值”），后面也一样。 在你高中甚至更早的时候，你熟悉均值的定义。

定义：数值集合的均值是集合中所有元素的总和，除以集合中元素的数量。

`np.average` 和 `np.mean` 方法返回数组的均值。

```
not_symmetric = make_array(2, 3, 3, 9)
np.average(not_symmetric)
4.25
np.mean(not_symmetric)
4.25
```

基本性质

上面的定义和例子指出了均值的一些性质。

- 它不一定是集合中的一个元素。
- 即使集合的所有元素都是整数，也不一定是整数。
- 它在集合的最小值和最大值之间。
- 它不一定在两个极值的正中间；集合中一半的元素并不总是大于均值。
- 如果集合含有一个变量的值，以指定单位测量，则均值也具有相同的单位。

我们现在将研究一些其他性质，它有助于理解均值，并与其他统计量相关。

均值是个“平滑器”

你可以将均值视为“均衡”或“平滑”操作。 例如，将上面的 `not_symmetric` 中的条目设想为四个不同人的口袋中的美元。 为此，你先把所有的钱都放进一个大袋子，然后平均分配给四个人。 最开始，他们在口袋中装了不同数量的钱（2 美元，3 美元，3 美元和9 美元），但现在每个人都有平均数量 4.25 美元。

均值的性质

如果一个集合只包含 1 和 0，那么集合的总和就是集合中 1 的数量，集合的均值就是 1 的比例。

```
zero_one = make_array(1, 1, 1, 0)
sum(zero_one)
3
np.mean(zero_one)
0.75
```

捏可以将 1 替换为布尔值 `True` ，0 替换为 `False` 。

```
np.mean(make_array(True, True, True, False))
0.75
```

因为比例是均值的一个特例，随机样本均值的结果也适用于随机样本比例。

均值和直方图

集合 `{2, 3, 3, 9}` 的平均值是 4.25，这不是数据的“正中间的点”。 那么这是什么意思？

为了了解它，请注意，平均值可以用不同的方式计算。

$$\begin{aligned}\text{mean} ~ &= ~ 4.25 \\ &= ~ \frac{2 + 3 + 3 + 9}{4} \\ &= ~ 2 \cdot \frac{1}{4} ~~ + ~~ 3 \cdot \frac{1}{4} ~~ + ~~ 3 \cdot \frac{1}{4} ~~ + ~~ 9 \cdot \frac{1}{4} \\ &= ~ 2 \cdot \frac{1}{4} ~~ + ~~ 3 \cdot \frac{2}{4} ~~ + ~~ 9 \cdot \frac{1}{4} \\ &= ~ 2 \cdot 0.25 ~~ + ~~ 3 \cdot 0.5 ~~ + ~~ 9 \cdot 0.25\end{aligned}$$

最后一个表达式就是一个普遍事实的例子：当我们计算平均值时，集合中的每个不同的值都由它在集合中出现的时间比例加权。

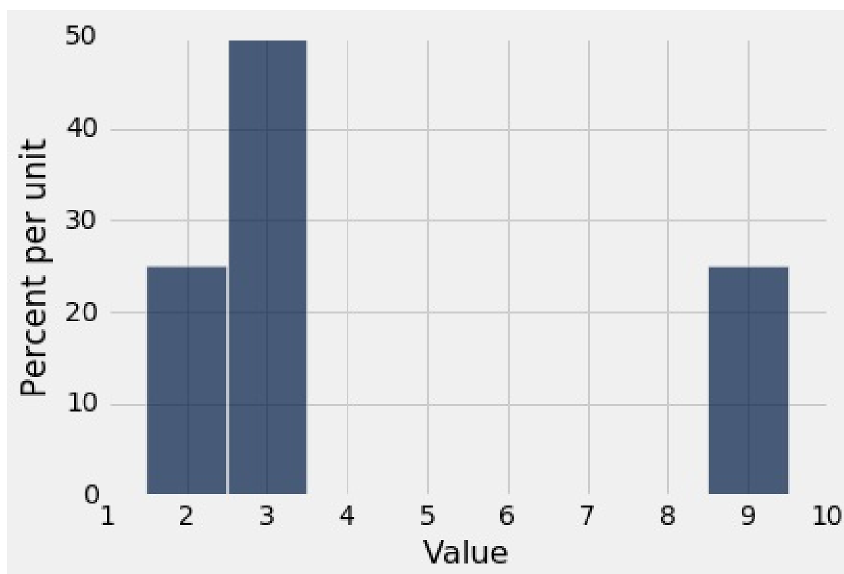
这有一个重要的结果。 集合的平均值仅取决于不同的值及其比例，而不取决于集合中元素的数量。 换句话说，集合的平均值仅取决于集合中值的分布。

因此，如果两个集合具有相同的分布，则它们具有相同的均值。

例如，这里是另一个集合，它的分布与 `not_symmetric` 相同，因此均值也相同。

```
not_symmetric
array([2, 3, 3, 9])
same_distribution = make_array(2, 2, 3, 3, 3, 3, 9, 9)
np.mean(same_distribution)
4.25
```

均值是分布直方图的物理属性。这里是 `not_symmetric` 的分布直方图，或者等价的 `same_distribution` 的分布直方图。

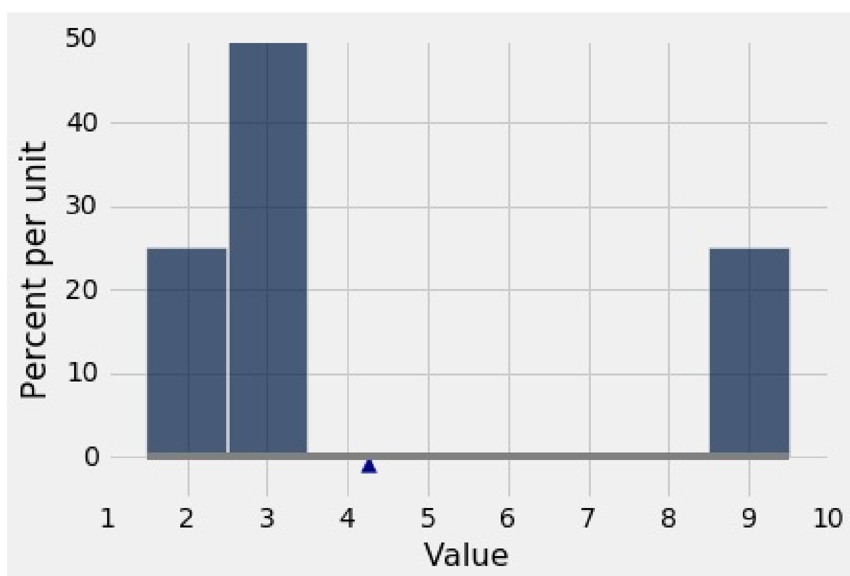


想象一下，直方图是由纸板组成的图形，它附着在一条线上，线沿着横轴延伸。并且，将这些条形想象为附加在值 2, 3 和 9 上的权重。假设你尝试在线上的某个点平衡这个图形。如果该点接近 2，图形就向右倾斜。如果该点接近 9，则图形就向左倾斜。之间的某个地方是这个数字取得平衡的点。这个点是 4.25，就是均值。

均值是直方图的重心或平衡点。

为了理解这是为什么，了解一些物理会有帮助。重心的计算与我们计算平均值的方法完全相同，通过将不同值按它们比例加权。

因为均值是一个平衡点，有时在直方图的底部显示为一个支点或三角形。



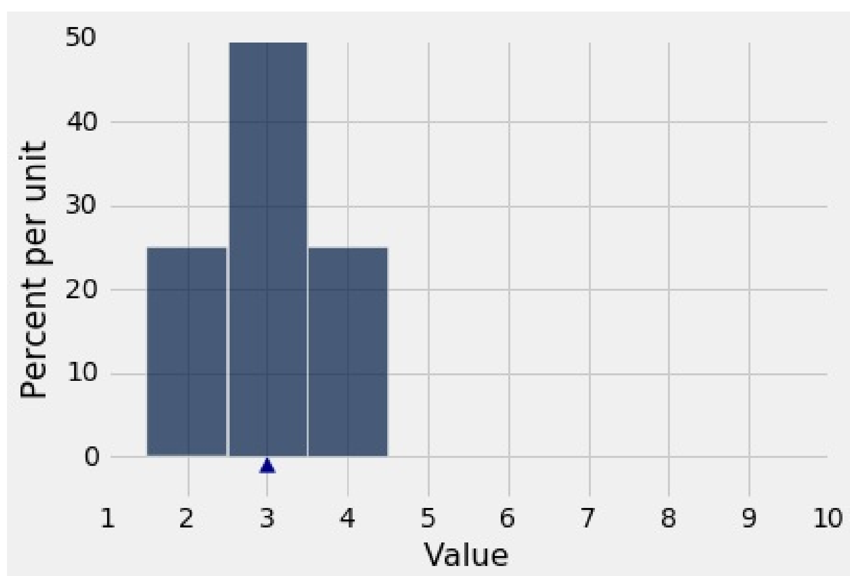
均值和中位数

如果一个学生的考试成绩低于平均水平，这是否意味着该学生在该考试中处于后一半？

对于学生来说，回答是“不一定”。 原因与直方图的平衡点即均值，和数据的“中间点”即中位数之间的关系有关。

通过这个关系很容易看到一个简单的例子。 这里是数组 `symmetric` 的集合 `{2, 3, 3, 4}` 的直方图。 分布对称于 3。均值和中位数都等于 3。

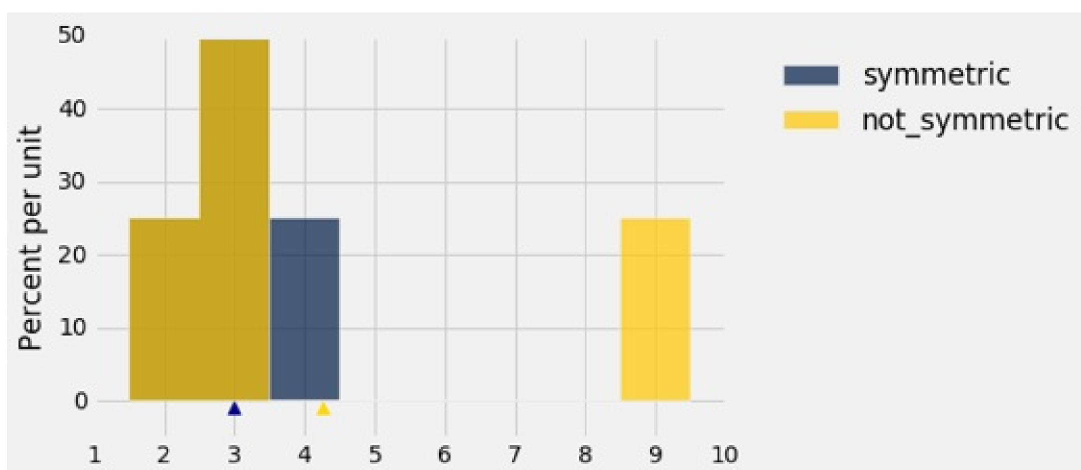
```
symmetric = make_array(2, 3, 3, 4)
```




```
np.mean(symmetric)
3.0
percentile(50, symmetric)
3
```

一般来说，对于对称分布，均值和中位数是相等的。

如果分布不对称呢？我们来比较 `symmetric` 和 `not_symmetric` 。



蓝色直方图表示原始的 `symmetric` 分布。 `not_symmetric` 的金色从左端起始，和蓝色一样，但是最右边的条形到了数值 9。棕色部分是两个直方图重叠的位置。

蓝色分布的中位数和均值都等于 3。金色分布的中值也等于 3，尽管右半部分与左边的分布不同。

但金色分布的平均值不是 3：金色直方图在 3 时不平衡。平衡点已经向右移动到 4.25。

在金色分布中，4 个条目中有 3 个（75%）低于平均水平。 因此，低于平均分的学生可以放心。 他或她可能是班上的大多数人。

一般来说，如果直方图的一边有尾巴（整数属于是“偏斜的”），那么平均值就会从中间拉到尾巴的方向。

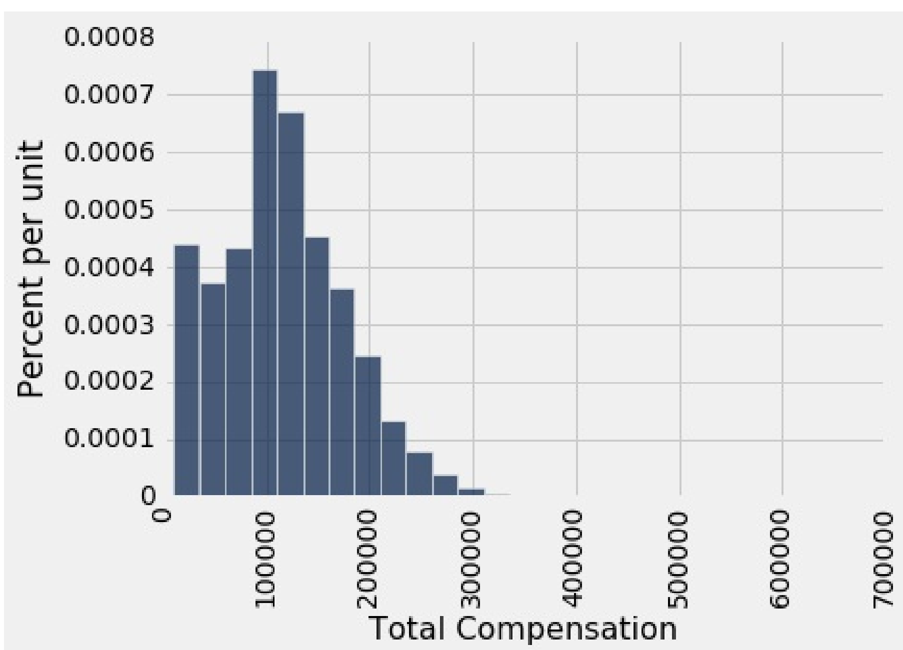
示例

`sf2015` 表包含 2015 年旧金山城市员工的薪水和福利数据。与以前一样，我们将我们的分析仅限于那些等价于至少就业半年的人。

```
sf2015 = Table.read_table('san_francisco_2015.csv').where('Salaries', are.above(10000)
)
```

我们前面看到了，最高薪资高于 60 万美元，但绝大多数雇员的薪资低于 30 万美元。

```
sf2015.select('Total Compensation').hist(bins = np.arange(10000, 700000, 25000))
```



这个直方图向右偏斜；它的右侧有个尾巴。

平均值拉向了尾巴的方向。 所以我们预计平均薪酬会比中位数大，事实确实如此。

```
compensation = sf2015.column('Total Compensation')
percentile(50, compensation)
110305.78999999999
np.mean(compensation)
114725.98411824222
```

大量总体的收入分布往往是右偏的。 当总体的大部分收入中到低，但很小一部分收入很高时，直方图的右侧有条细长的尾巴。

平均收入受这条尾巴的影响：尾巴向右延伸得越远，平均值就越大。 但中位数不受分布极值的影响。 这就是经济学家经常用收入分布的中位数来代替平均值的原因。

可变性

平均值告诉我们直方图平衡的位置。 但是在我们所看到的几乎所有的直方图中，值都位于均值的两边。 他们距离均值有多远？ 为了回答这个问题，我们将开发一个关于均值的可变性度量。

我们首先描述如何计算度量值。 然后我们就会明白，为什么这是很好的计算方法。

距离均值的偏差的大致大小

为了简单起见，我们将在简单数组`any_numbers`的上下文中开始计算，它由四个值组成。 你将会看到，我们的方法非常易于扩展到任何其他数组。

```
any_numbers = make_array(1, 2, 2, 10)
```

我们的目标是，大致衡量这些数值离他们的平均水平有多远。 为了实现它，我们首先需要均值：

```
# Step 1. The average.

mean = np.mean(any_numbers)
mean
3.75
```

接下来，我们来看看每个数值离均值有多远。 这些被称为到均值的偏差。“到均值的偏差”只是每个值减去平均值。 `calculation_steps`表显示了结果。

```
# Step 2. The deviations from average.

deviations = any_numbers - mean
calculation_steps = Table().with_columns(
        'Value', any_numbers,
        'Deviation from Average', deviations
        )
calculation_steps
```

Value	Deviation from Average
1	-2.75
2	-1.75
2	-1.75
10	6.25

一些偏差是负的；它们对应于低于均值的值。 正的偏差对应于高于平均值的值。

要计算偏差有多大，计算偏差的平均值是很自然的。 但是当所有的偏差加在一起的时候，会发生一些有趣的事：

```
sum(deviations)
0.0
```

正的偏差正好和负的偏差抵消。 无论列表的直方图是什么样子，所有的数字列表都是如此：到均值的偏差总和为零。

由于偏差的总和为零，偏差的均值也将为零：

```
np.mean(deviations)
0.0
```

因此，偏差的均值不是偏差大小的有用度量。 我们真正想知道的是偏差有多大，不管它们是正的还是负的。 所以我们需要一种方法来消除偏差的符号。

有两种历史悠久的丢掉符号的方式：绝对值和平方。 事实证明，采用平方会构建一个度量，带有非常强大的性质，其中一些我们将在这个课程中学习。

所以让我们计算所有偏差的平方，来消除符号。 那么我们将计算平方的均值：

```
# Step 3. The squared deviations from average

squared_deviations = deviations ** 2
calculation_steps = calculation_steps.with_column(
    'Squared Deviations from Average', squared_deviations
    )
calculation_steps
```

Value	Deviation from Average	Squared Deviations from Average
1	-2.75	7.5625
2	-1.75	3.0625
2	-1.75	3.0625
10	6.25	39.0625

```
# Step 4. Variance = the mean squared deviation from average

variance = np.mean(squared_deviations)
variance
13.1875
```

方差：上面计算的偏差平方的均值称为方差。

虽然方差确实给了我们延展度的概念，但它和原始变量不是一个量纲，因为它的单位是原始变量的平方。 这使得解释非常困难。

所以我们通过计算方差的算术平方根的来返回原来的量纲：

```
# Step 5.
# Standard Deviation:    root mean squared deviation from average
# Steps of calculation:   5    4      3       2             1

sd = variance ** 0.5
sd
3.6314597615834874
```

标准差

我们刚计算出来的数量叫做列表的标准差，简写为 SD。 它大致衡量列表中的数字与其平均水平的差距。

定义：列表的 SD 定义为方差（偏差平方的均值）的算术平方根。这很拗口。 但是从左到右阅读，你需要执行一系列的步骤的计算。

计算：上述五个步骤会产生 SD。 你还可以使用函数 `np.std` 来计算数组中值的标准差：

```
np.std(any_numbers)
3.6314597615834874
```

> 译者注：写在一起就是 `np.mean((arr - arr.mean()) ** 2) ** 0.5`。

使用 SD

要看看我们可以从SD中学到什么，让我们转向一个比 `any_numbers` 更有趣的数据集。 `nba13` 表包含了 2013 年 NBA 的球员数据。对于每个球员来说，表格中记录了球员通常的位置，他的身高（英寸），体重（磅）和年龄。

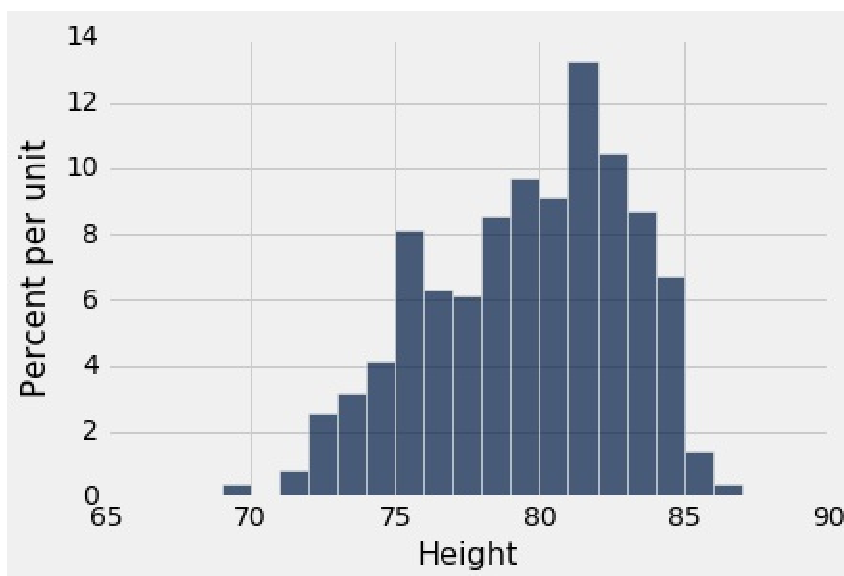
```
nba13 = Table.read_table('nba2013.csv')
nba13
```

Name	Position	Height	Weight	Age in 2013
DeQuan Jones	Guard	80	221	23
Darius Miller	Guard	80	235	23
Trevor Ariza	Guard	80	210	28
James Jones	Guard	80	215	32
Wesley Johnson	Guard	79	215	26
Klay Thompson	Guard	79	205	23
Thabo Sefolosha	Guard	79	215	29
Chase Budinger	Guard	79	218	25
Kevin Martin	Guard	79	185	30
Evan Fournier	Guard	79	206	20

（省略了 495 行）

这里是球员身高的直方图。

```
nba13.select('Height').hist(bins=np.arange(68, 88, 1))
```



NBA 球员身材高大并不奇怪！他们的平均身高只有 79 英寸（6'7"），比美国男子的平均身高高出 10 英寸。

```
mean_height = np.mean(nba13.column('Height'))
mean_height
79.06534653465347
```

球员的身高距离平均有多远？这通过身高的 SD 来测量，大约是 3.45 英寸。

```
sd_height = np.std(nba13.column('Height'))
sd_height
3.4505971830275546
```

俄克拉荷马雷霆的高个中锋哈希姆·塔比特（Hasheem Thabeet）是最高的球员，身高 87 英寸。

```
nba13.sort('Height', descending=True).show(3)
```

Name	Position	Height	Weight	Age in 2013
Hasheem Thabeet	Center	87	263	26
Roy Hibbert	Center	86	278	26
Tyson Chandler	Center	85	235	30

（省略了 502 行）

Thabeet 比平均身高高了大约 8 英寸。

```
87 - mean_height
7.9346534653465284
```

这个就是距离均值的偏差，大约是 2.3 乘标准差。

```
(87 - mean_height)/sd_height
2.2995015194397923
```

换句话说，最高球员的身高比均值高了 2.3 个 SD。

以赛亚·托马斯（Isaiah Thomas）身高 69 英寸，是 2013 年 NBA 最矮的球员之一。他的身高比均值低了 2.9 个 SD。

```
nba13.sort('Height').show(3)
```

Name	Position	Height	Weight	Age in 2013
Isaiah Thomas	Guard	69	185	24
Nate Robinson	Guard	69	180	29
John Lucas III	Guard	71	157	30

（省略了 502 行）

```
(69 - mean_height)/sd_height
-2.9169868288775844
```

我们观察到，最高和最矮的球员都距离平均身高只有几个标准差。 这是例子，说明了为什么 SD 是延展度的有效度量。无论直方图的形状如何，平均值和 SD 一起告诉你很多东西，关于直方图在数轴上的位置。

使用 SD 度量延展度的最主要原因

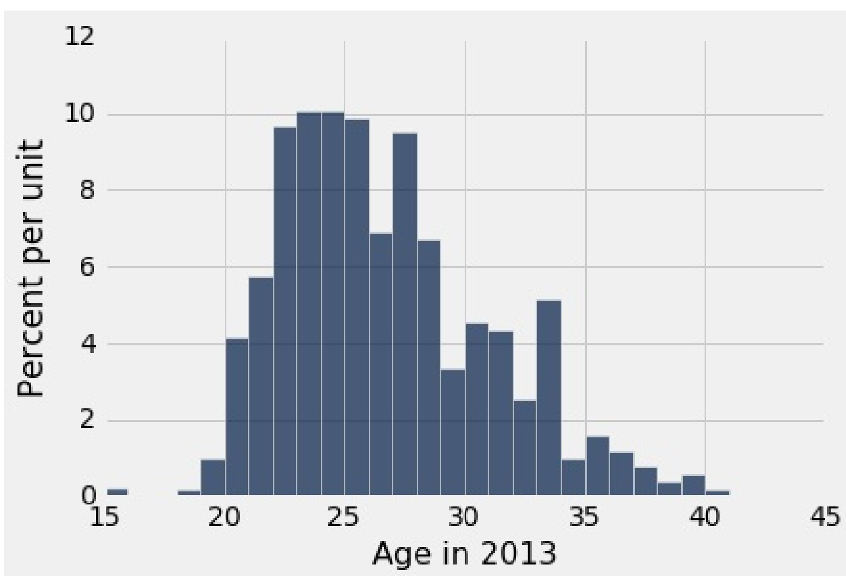
非正式声明：在所有的数值数据集中，大部分条目都在“均值上下几个标准差”的范围内。

现在，先克制住自己，不要了解“散”，“少”等模糊词的确切含义。 我们将在本节的后面进行详细说明。 我们仅仅在更多示例的背景下研究这个陈述。

我们已经看到，所有 NBA 球员的身高都在“均值上下几个标准差”的范围内。

那年龄呢？ 这里是分布的直方图，以及年龄的平均值和标准差。

```
nba13.select('Age in 2013').hist(bins=np.arange(15, 45, 1))
```




```
ages = nba13.column('Age in 2013')
mean_age = np.mean(ages)
sd_age = np.std(ages)
mean_age, sd_age
(26.19009900990099, 4.3212004417203067)
```

平均年龄只有 26 岁，标准差大约是 4.3 岁。

年龄与均值相差多远？ 就像我们对身高所做的那样，让我们看看两个年龄的极端值。

Juwan Howard 是年龄最大的球员 40 岁。

```
nba13.sort('Age in 2013', descending=True).show(3)
```

Name	Position	Height	Weight	Age in 2013
Juwan Howard	Forward	81	250	40
Marcus Camby	Center	83	235	39
Derek Fisher	Guard	73	210	39

（省略了 502 行）

Howard 的年龄比均值高了 3.2 个标准差。

```
(40 - mean_age)/sd_age
3.1958482778922357
```

年龄最小的是 15 岁的 Jarvis Varnado，他当年在迈阿密热火队（Miami Heat）夺得了 NBA 总冠军。 他的年龄比均值低了 2.6 个标准差。

```
nba13.sort('Age in 2013').show(3)
```

Name	Position	Height	Weight	Age in 2013
Jarvis Varnado	Forward	81	230	15
Giannis Antetokounmpo	Forward	81	205	18
Sergey Karasev	Guard	79	197	19

（省略了 502 行）

```
(15 - mean_age)/sd_age
-2.5895811038670811
```

对于高度和年龄，我们观察到的东西非常普遍。 对于所有列表，大部分条目都不超过平均值 2 或 3 个标准差。

切比雪夫边界

俄罗斯数学家切比雪夫（Pafnuty Chebychev，1821-1894）证明了这个结论，使我们的粗略陈述更加精确。

对于所有列表和所有数字 `z`，“均值上下 `z` 个标准差”范围内的条目比例至少为 $1 - \frac{1}{z^2}$。

值得注意的是，结果给出了一个界限，而不是一个确切的数值或近似值。

是什么让结果变得强大，对于所有列表来说都是这样呢 - 所有的分布，无论多么不规则？

具体来说，对于每个列表：

在“均值上下两个标准差”范围内的比例至少是 `1 - 1/4 = 0.75`

在“均值上下三个标准差”范围内的比例至少为 `1 - 1/9 ≈ 0.89`

在“均值上下 4.5 个标准差”范围内的比例至少为 `1 - 1/4.5^2 ≈ 0.95`

如上所述，切比雪夫的结果给出了一个下界，而不是一个确切的答案或近似值。例如，“均值上下两个标准差”范围内的条目百分比可能比 75% 大得多。但它不会更小。

标准单位

在上面的计算中，`z` 的数量是标准单位，高于平均水平的标准差的数量。

标准单位的某些值是负值，对应于低于均值的原始值。 标准单位的其他是正值。 但是无论列表的分布如何，切比雪夫边界意味着标准单位一般在 `(-5, 5)` 范围内。

要将一个值转换为标准单位，首先要求出距离平均值有多远，然后将该偏差与标准差比较。

$$z = \frac{\mbox{value} - \mbox{average}}{\mbox{SD}}$$

我们将会看到，标准单位经常用于数据分析。 所以定义一个函数，将数值的数组转换为标准单位是很有用的。

```
def standard_units(numbers_array):
    "Convert any array of numbers to standard units."
    return (numbers_array - np.mean(numbers_array))/np.std(numbers_array)
```

示例

我们在前面的章节中看到， `united` 表包含了 `Delay` 列，包括 2015 年夏天联合航空数千航班的起飞延误时间，以分钟为单位。我们将创建一个名为 `Delay (Standard Units)` 的新列， 通过将函数 `standard_units` 应用于 `Delay` 列。 这使我们可以看到所有延误时间（分钟）以及标准单位的相应值。

```
united = Table.read_table('united_summer2015.csv')
united = united.with_column(
    'Delay (Standard Units)', standard_units(united.column('Delay'))
)
united
```

Date	Flight Number	Destination	Delay	Delay (Standard Units)
6/1/15	73	HNL	257	6.08766
6/1/15	217	EWR	28	0.287279
6/1/15	237	STL	-3	-0.497924
6/1/15	250	SAN	0	-0.421937
6/1/15	267	PHL	64	1.19913
6/1/15	273	SEA	-6	-0.573912
6/1/15	278	SEA	-8	-0.62457
6/1/15	292	EWR	12	-0.117987
6/1/15	300	HNL	20	0.0846461
6/1/15	317	IND	-10	-0.675228

（省略了 13815 行）

我们看到的标准单位与我们根据切比雪夫边界的预期一致。 大部分都是相当小的值；只有一个大于 6。

但是，当我们将延误时间从高到低排序时，会发生一些惊人的事情。 我们看到的标准单位是非常高的！

```
united.sort('Delay', descending=True)
```

Date	Flight Number	Destination	Delay	Delay (Standard Units)
6/21/15	1964	SEA	580	14.269
6/22/15	300	HNL	537	13.1798
6/21/15	1149	IAD	508	12.4453
6/20/15	353	ORD	505	12.3693
8/23/15	1589	ORD	458	11.1788
7/23/15	1960	LAX	438	10.6722
6/23/15	1606	ORD	430	10.4696
6/4/15	1743	LAX	408	9.91236
6/17/15	1122	HNL	405	9.83637
7/27/15	572	ORD	385	9.32979

（省略了 13815 行）

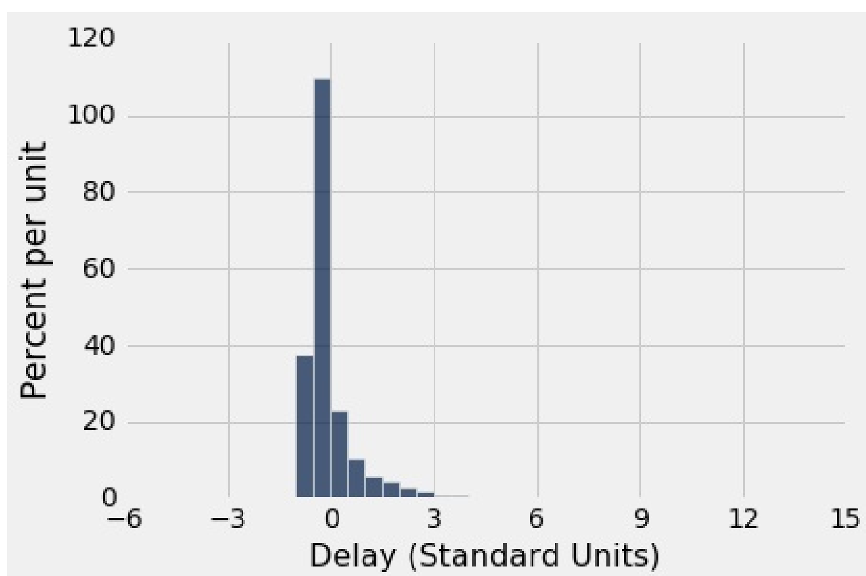
这表明，数据有可能高于均值很多个标准差（对于延误了 10 个小时的航班）。 延误的最高值超过 14 个标准单位。

然而，这些极端值的比例很小，切比雪夫边界仍然是真的。 例如，让我们计算在“均值上下三个标准差”范围内的延误百分比。 这与标准单位在 `(-3, 3)` 范围内的时间百分比相同。 这大约是 98%，计算在下面，和切比雪夫边界“至少 89%”一致。

```
within_3_sd = united.where('Delay (Standard Units)', are.between(-3, 3))
within_3_sd.num_rows/united.num_rows
0.9790235081374322
```

延误时间的直方图如下所示，横轴以标准单位表示。 从上表中可以看出，右边的尾巴一直延伸到 `z = 14.27` 个标准单位（580 分钟）。 在 `z = -3` 到 `z = 3` 范围外的直方图面积大约是 2%，加起来非常小，在直方图中几乎不可见。

```
united.hist('Delay (Standard Units)', bins=np.arange(-5, 15.5, 0.5))
plots.xticks(np.arange(-6, 17, 3));
```



标准差和正态曲线

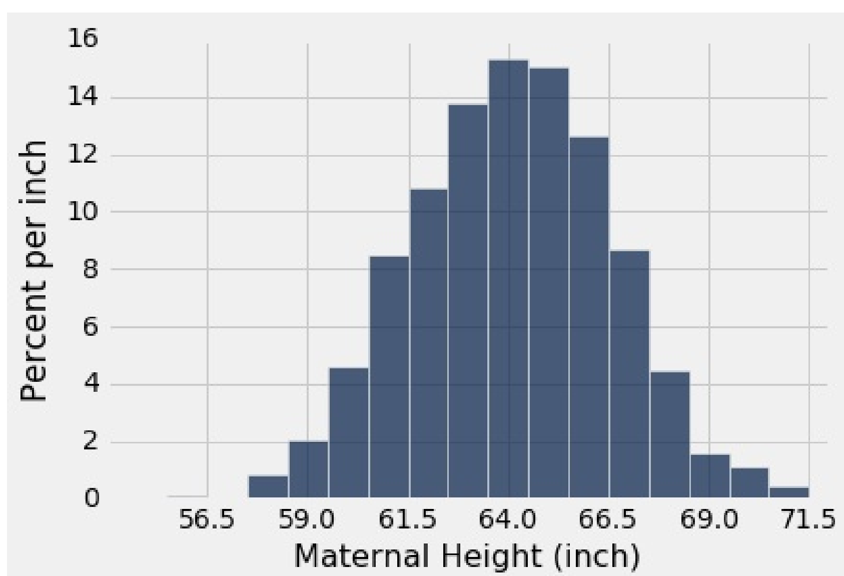
我们知道均值是直方图的平衡点。 标准差与平均值不同，通常不容易通过查看直方图来识别。

然而，有一种分布形状，它的标准差与平均值几乎一样清晰可辨。 这是钟形分布。 本节将查看该形状，因为它经常出现在概率直方图中，也出现在一些数据的直方图中。

数据的大致钟形的直方图

让我们看看母亲的身高分布，它们在我们熟悉的 1174 对母亲和新生儿的样本中。母亲的平均身高为 64 英寸，SD 为 2.5 英寸。 与篮球运动员的身高不同，母亲身高关于钟形曲线中的平均值对称分布。

```
baby = Table.read_table('baby.csv')
heights = baby.column('Maternal Height')
mean_height = np.round(np.mean(heights), 1)
mean_height
64.0
sd_height = np.round(np.std(heights), 1)
sd_height
2.5
baby.hist('Maternal Height', bins=np.arange(55.5, 72.5, 1), unit='inch')
positions = np.arange(-3, 3.1, 1)*sd_height + mean_height
plots.xticks(positions);
```



上面单元格中的最后两行代码更改了横轴的标签。 现在，对于 `z=0, ±1, ±2, ±3`，标签对应于“标签上下 `z` 个标准差”。 由于分布的形状，“中心”具有明确的含义，在 64 处清晰可见。

如何定位钟形曲线上的 SD

要看 SD 如何与曲线相关，请从曲线顶部开始，向右看。 请注意，曲线有一个地方，从看起来像“倒扣的杯子”，变为“朝右的杯子”。 在形式上，曲线有一个拐点。 这个点高于均值一个 SD。 这是 `z = 1` 的点，即“均值加一个标准差”，为 66.5 英寸。

在均值的左边也对称，拐点在 `z = -1` 处，也就是“均值减一个标准差”，为 61.5 英寸。

一般来说，对于钟形分布，SD 是均值和任一侧的拐点之间的距离。

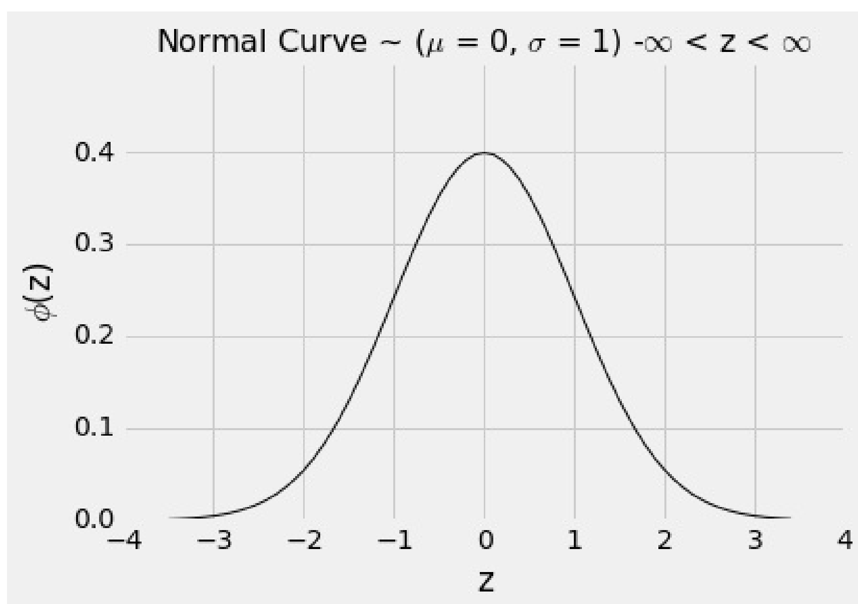
标准正态曲线

除了轴上的标签，我们所看到的所有钟形直方图，看起来基本相同。 的确，通过适当地重新标记坐标轴，从所有这些曲线中，实际上只能绘制一条曲线。

为了绘制这条基本曲线，我们将使用标准单位，我们可以将每个列表转换成它。所得到的曲线因此被称为标准正态曲线。

标准正态曲线的方程令人印象深刻。 但是现在，最好把它看作是变量直方图的平滑轮廓，变量以标准单位测量并具有钟形分布。

$$\phi(z) = \frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}z^2}, \quad -\infty < z < \infty$$



与往常一样，当你检查新的直方图时，首先查看横轴。在标准正态曲线的横轴上，这些值是标准单位。

这里是曲线的一些属性。有些是通过观察显而易见的，有些则需要大量的数学才能建立起来。

曲线下面的总面积是1.所以你可以把它看作是绘制为密度标度的直方图。

曲线是对称的。所以如果一个变量具有这个分布，它的平均值和中位数都是 0。

曲线的拐点在 -1 和 +1 处。

如果一个变量具有这种分布，那么它的 SD 是 1。正态曲线是 SD 清晰可辨的极少数分布之一。

由于我们将曲线视为平滑的直方图，因此我们希望用曲线下方的面积来表示数据总量的比例。

平滑曲线下的面积通常是通过微积分来计算的，使用一种称为积分的方法。然而，一个数学的事实是，标准的正态曲线不能通过任何微积分方式来积分。

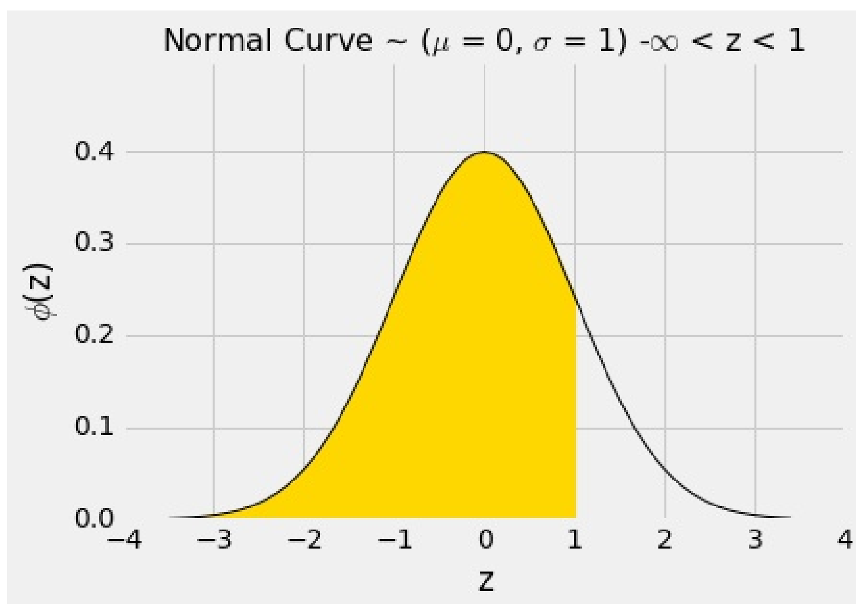
因此，曲线下方的面积必须近似。这就是几乎所有的统计教科书，都带有曲线下方的面积的原因。这也是所有统计系统，包括 Python 模块在内，都包含提供这些面积的优秀近似的方法的原因。

```
from scipy import stats
```

标准正态的累积分布函数（CDF）

用于求出正态曲线下的面积的基本函数是 `stats.norm.cdf` 。 它接受一个数值参数，并返回曲线下，该数值的左侧的所有面积。 它在形式上被称为标准正态曲线的“累积分布函数”。 在口语里缩写为 CDF。

让我们使用这个函数来求出标准正态曲线下， `z=1` 左侧的面积。

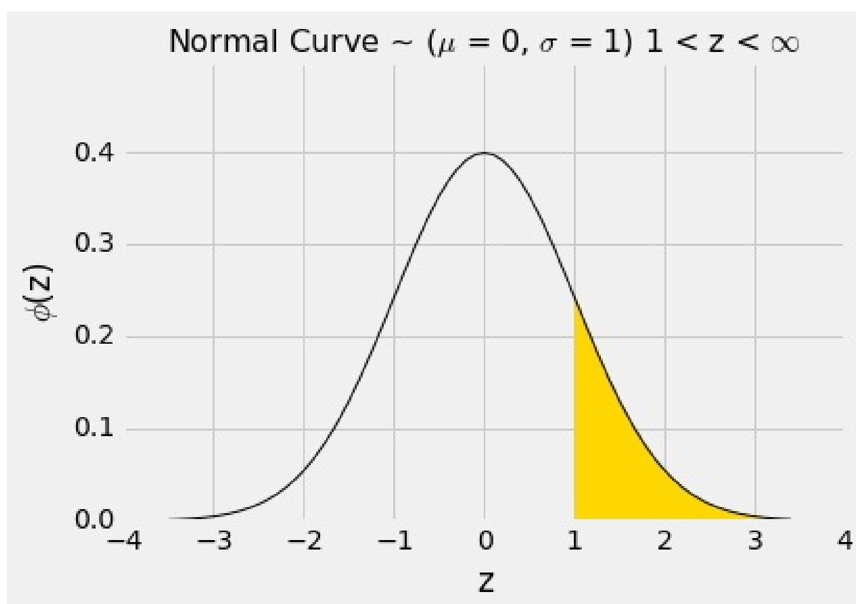


阴影区域的数值可以通过调用 `stats.norm.cdf` 来求出。

```
stats.norm.cdf(1)
0.84134474606854293
```

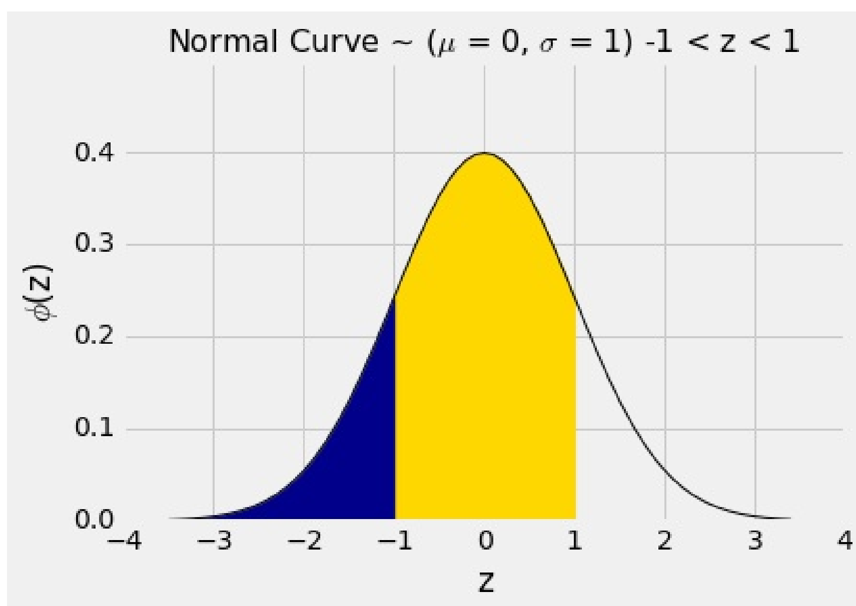
这大概是 84%。 现在我们可以使用曲线的对称性，以及曲线下面的总面积为 1 事实，来求出其他面积。

`z = 1` 右侧的面积大概是 `100% - 84% = 16%` 。




```
1 - stats.norm.cdf(1)
0.15865525393145707
```

`z = -1` 和 `z = 1` 之间的面积可以用几种不同的方式来计算。 它是下面的曲线下方的金色区域。

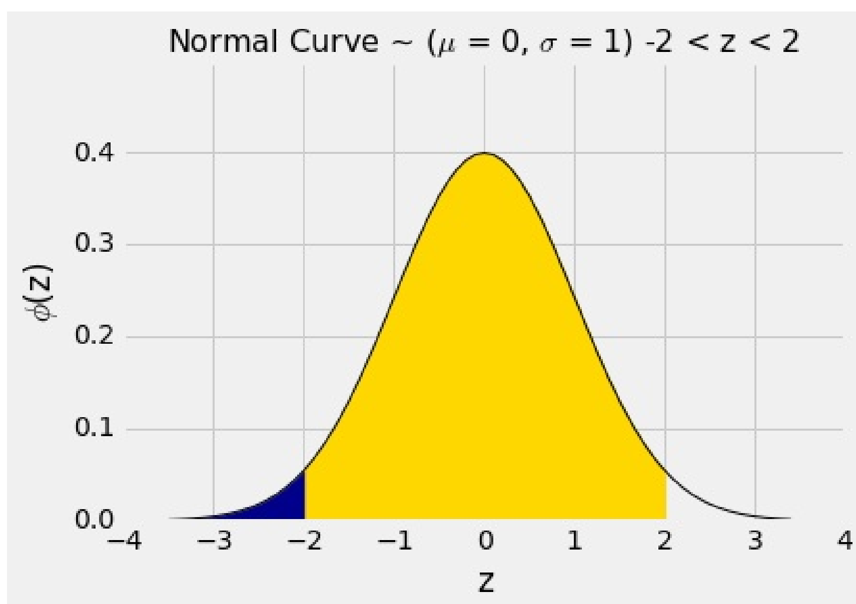


例如，我们可以将面积计算为“ `100% -` 两个相等的尾巴”，结果大致是 `100% - 2X16% = 68%` 。

或者我们可以注意到， `z = 1` 和 `z = -1` 之间的区域等于 `z = 1` 左边的所有区域，减去 `z = -1` 左边的所有区域。

```
stats.norm.cdf(1) - stats.norm.cdf(-1)
0.6826894921370859
```

通过类似的计算，我们看到 `-2` 和 `2` 之间的区域大约是 95%。




```
stats.norm.cdf(2) - stats.norm.cdf(-2)
0.9544997361036416
```

换句话说，如果一个直方图大致是钟形，那么在“均值上下两个标准差”范围内的数据比例大约是 95%。

这比切比雪夫的下界 75% 还要多。 切比雪夫边界较弱，因为它必须适用于所有的分布。 如果我们知道一个分布是正态的，那么我们就有很好的比例近似，而不仅仅是边界。

下表比较了我们对所有分布和正态分布的了解。 请注意，当 `z = 1` 时，切比雪夫的边界是正确的，但没有启发性。

Percent in Range	All Distributions: Bound	Normal Distribution: Approximation
均值上下一个标准差	至少 0%	约 68%
均值上下两个标准差	至少 75%	约 95%
均值上下三个标准差	至少 88.888...%	约 99.73%

中心极限定律

我们在本课程中看到的很少数据直方图是钟形的。 当我们遇到一个钟形的分布时，它几乎总是一个基于随机样本的统计量的经验直方图。

下面的例子显示了两个非常不同的情况，其中在这样的直方图中出现了近似的钟形。

轮盘赌的净收益

在前面的章节中，如果我们在轮盘的不同轮次上重复下相同的赌注，那么我们所花费的总金额的粗略形状就会成为钟形。

```
wheel
```

Pocket	Color
0	green
00	green
1	red
2	black
3	red
4	black
5	red
6	black
7	red
8	black

（省略了 28 行）

回想一下，红色的下注返回相等的钱，1 比 1。我们定义的函数 `red_winnings` 返回对红色下注一美元的净收益。具体来说，该函数将颜色作为参数，如果颜色为红色，则返回 1。 对于所有其他颜色，它返回 -1。

```
def red_winnings(color):
    if color == 'red':
        return 1
    else:
        return -1
```

`red` 表展示了红色情况下，每个口袋的奖金。

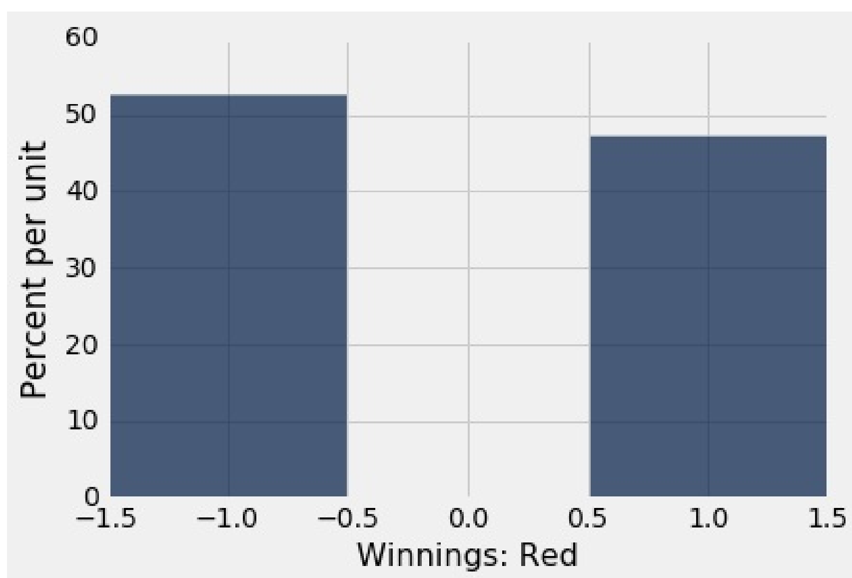
```
red = wheel.with_column(
    'Winnings: Red', wheel.apply(red_winnings, 'Color')
    )
red
```

Pocket	Color	Winnings: Red
0	green	-1
00	green	-1
1	red	1
2	black	-1
3	red	1
4	black	-1
5	red	1
6	black	-1
7	red	1
8	black	-1

（省略了 28 行）

你在赌注上的净收益 `Winnings: Red` 的随机抽样。 有 1/18 的几率赚一美元，20/38 的几率损失一美元。 这个概率分布显示在下面的直方图中。

```
red.select('Winnings: Red').hist(bins=np.arange(-1.5, 1.6, 1))
```



现在假设你多次对红色下注。 你的净收益将是来自上述分布的，多个带放回随机抽样的总和。

这将需要一些数学，来列出净收益的所有可能值，以及所有的记录。 我们不会那样做；相反，我们将通过模拟来逼近概率分布，就像我们在这个过程中一直做的那样。

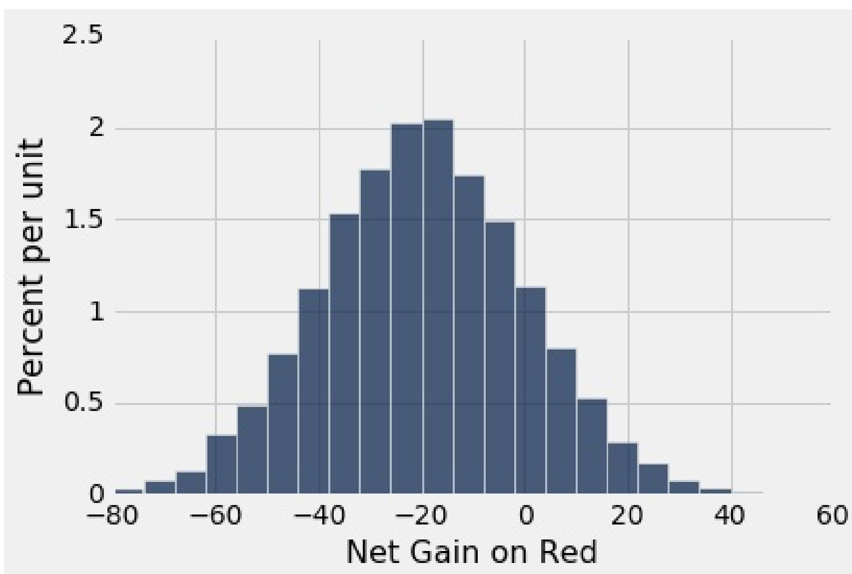
下面的代码模拟你的净收益，如果你在轮盘赌的 400 个不同的轮次中，对红色下注一美元。

```
num_bets = 400
repetitions = 10000

net_gain_red = make_array()

for i in np.arange(repetitions):
    spins = red.sample(num_bets)
    new_net_gain_red = spins.column('Winnings: Red').sum()
    net_gain_red = np.append(net_gain_red, new_net_gain_red)


results = Table().with_column(
    'Net Gain on Red', net_gain_red
    )
results.hist(bins=np.arange(-80, 50, 6))
```



这是一个大致钟形的直方图，即使我们正在绘制的分布并不是钟形。

中心。分布集中在 `-$20` 附近。 要知道为什么，请注意，你的奖金在 18/38 左右的下注中为 1 美元，剩下的 20/38 则为负一美元。 所以每个一美元赌注的平均奖金大概是 -5.26 美分：

```
average_per_bet = 1*(18/38) + (-1)*(20/38)
average_per_bet
-0.05263157894736842
```

因此，在 400 次下注中，你预计净收益大约是 21 美元。

```
400 * average_per_bet
-21.052631578947366
```

为了确认，我们可以计算 10,000 次模拟净收益的平均值：

```
np.mean(results.column(0))
-20.8992
```

延展。让你的眼睛沿着曲线从中心开始，注意到拐点在 0 附近。在钟形曲线上，SD 是中心到拐点的距离。 中心大概是 -20 美元，这意味着分布的标准差大约是 20 美元。

在下一节中，我们将看到 20 美元是怎么来的。 现在，让我们通过简单计算 10,000 个模拟净收益的 SD 来证实我们的观察：

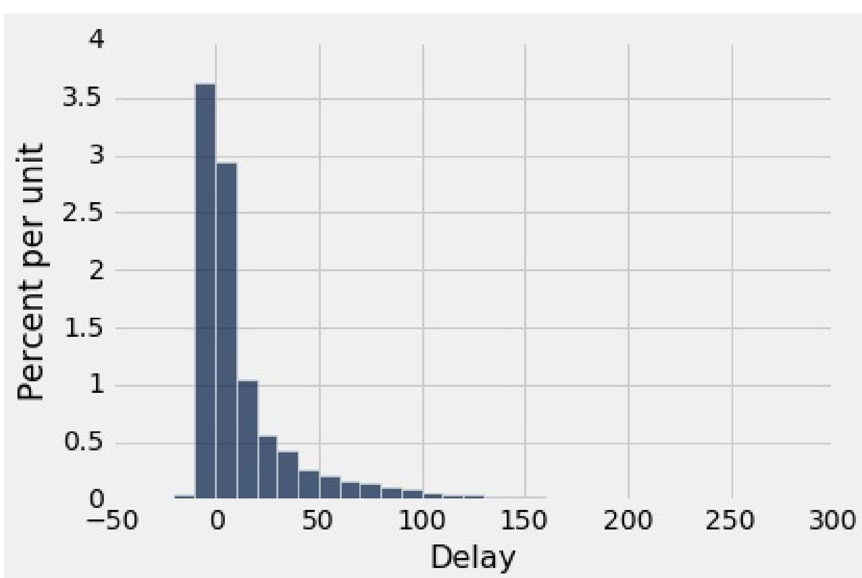
```
np.std(results.column(0))
20.043159415621083
```

总结。 400 次下注的净收益是每个单独赌注的 400 个奖金的总和。 这个总和的概率分布近似正态，我们可以近似它的均值和标准差。

平均航班延误

`united` 表包含 2015 年夏季旧金山机场出发的 13,825 个联合航空国内航班的出发延误数据。正如我们以前所见，延误的分布的右侧有着很长的尾巴。

```
united = Table.read_table('united_summer2015.csv')
united.select('Delay').hist(bins=np.arange(-20, 300, 10))
```



平均延误约为 16.6 分钟，SD 约为 39.5 分钟。 注意 SD 与平均值相比有多大。 但是右侧的较大偏差会产生影响，尽管它们在数据中占很小的比例。

```
mean_delay = np.mean(united.column('Delay'))
sd_delay = np.std(united.column('Delay'))

mean_delay, sd_delay
(16.658155515370705, 39.480199851609314)
```

现在假设我们随机抽取了 400 个延误。 如果你愿意，你可以无放回抽样，但是结果与放回抽样非常相似。 如果你从 13,825 个中无放回地抽取几百个，那么每当你抽出一个值时，几乎不会改变总体。

在样本中，平均延误会是多少？ 我们预计在 16 或 17 左右，因为这是总体的均值。 但可能会有些偏差。 让我们看看我们通过抽样得到了什么。 我们将处理 `delay` 表，仅包含延迟的列。

```
delay = united.select('Delay')
np.mean(delay.sample(400).column('Delay'))
16.68
```

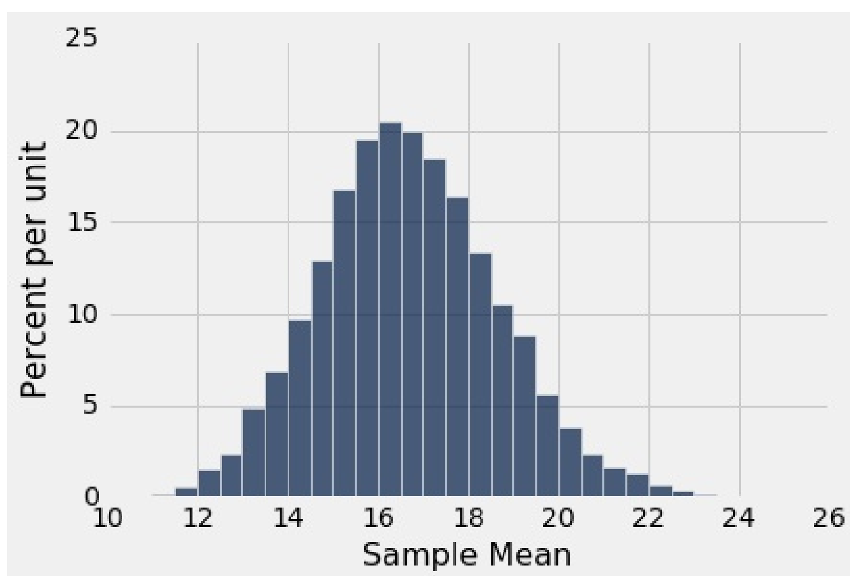
样本均值根据样本的出现方式而变化，因此我们将重复模拟抽样过程，并绘制样本均值的经验直方图。 这是样本均值的概率直方图的近似值。

```
sample_size = 400
repetitions = 10000

means = make_array()

for i in np.arange(repetitions):
    sample = delay.sample(sample_size)
    new_mean = np.mean(sample.column('Delay'))
    means = np.append(means, new_mean)

results = Table().with_column(
    'Sample Mean', means
)
results.hist(bins=np.arange(10, 25, 0.5))
```



即使我们从非常偏斜的分布抽样，我们再次看到了大致的钟形。 正如我们所期望的那样，这个钟形的中心在 16 到 17 之间。

中心极限定律

钟形出现在这样的环境中的原因，是一个概率理论的显着结果，称为中心极限定律。

中心极限定理表明，无论用于抽取样本的总体分布如何，带放回抽取的大型随机样本的总和或均值的概率分布大致是正态的。

我们在研究切比雪夫边界时指出，不管总体分布如何，结果都可以应用于随机样本，这非常强大，因为在数据科学中，我们很少知道总体的分布。

如果我们有一个大型随机样本，那么中心极限定理就能够在总体知识很少的情况下进行推理。 这就是它是统计推断领域的核心的原因。

紫色的花的分布

回忆孟德尔的豌豆植物的花朵颜色的概率模型。 该模型表明，植物的花朵颜色类似于来自`{紫色，紫色，紫色，白色}`的带放回随机抽样。

在植物的大型样本中，紫色的花约有多少比例？ 我们预计答案约为 0.75，模型中紫色的比例。 而且，由于比例是均值，中心极限定理表明，紫色的样本比例的分布大致是正态的。

我们可以通过模拟来确认。 我们来模拟 200 株植物样本中紫色的花的比例。

```
colors = make_array('Purple', 'Purple', 'Purple', 'White')

model = Table().with_column('Color', colors)

model
```

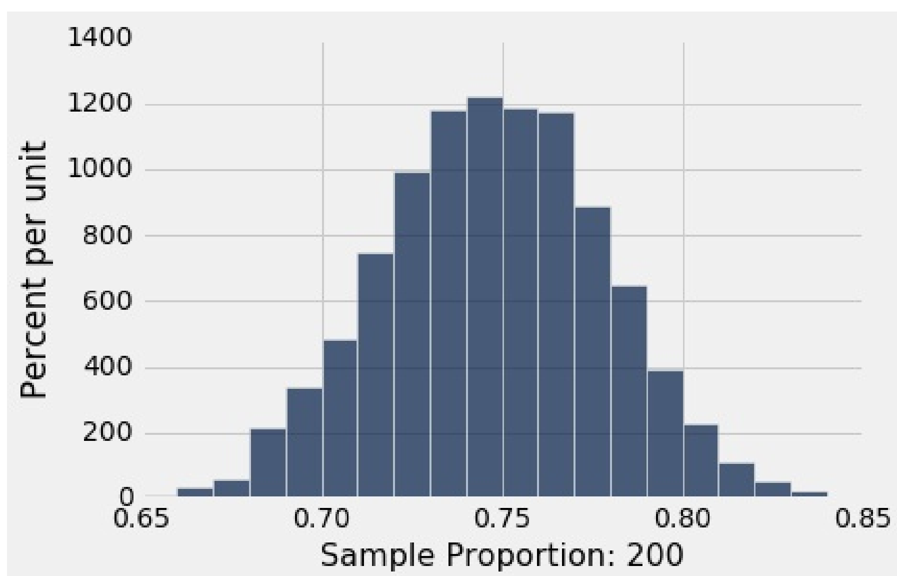
Color
Purple
Purple
Purple
White

```
props = make_array()

num_plants = 200
repetitions = 10000

for i in np.arange(repetitions):
    sample = model.sample(num_plants)
    new_prop = np.count_nonzero(sample.column('Color') == 'Purple')/num_plants
    props = np.append(props, new_prop)

results = Table().with_column('Sample Proportion: 200', props)
results.hist(bins=np.arange(0.65, 0.85, 0.01))
```



正如你所期望的那样，中央极限定理预测了，正态曲线再次集中于 0.75 左右。

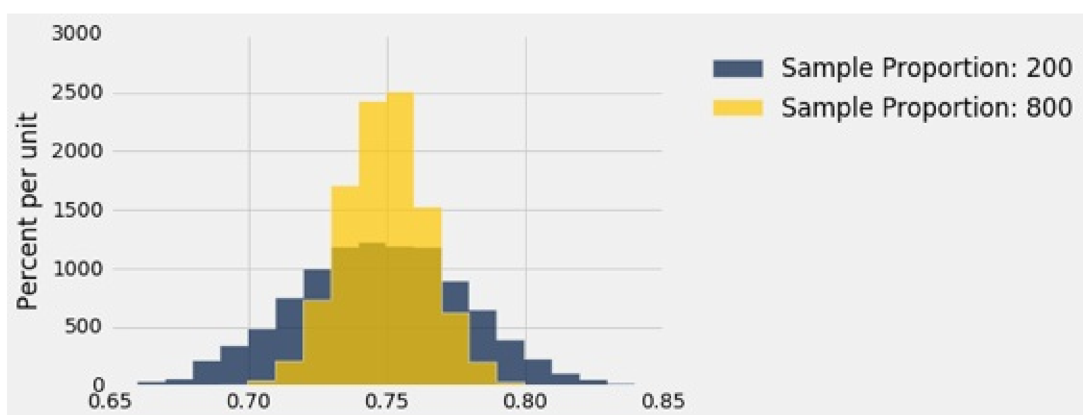
如果我们增加样本量，这个分布如何变化？让我们再次运行代码，样本量为 800 ，并将模拟结果收集在同一个表中，我们在里面收集了样本量为 200 的模拟结果。我们使重复次数与之前相同，以便两列具有相同的长度。

```
props2 = make_array()

num_plants = 800

for i in np.arange(repetitions):
    sample = model.sample(num_plants)
    new_prop = np.count_nonzero(sample.column('Color') == 'Purple')/num_plants
    props2 = np.append(props2, new_prop)

results = results.with_column('Sample Proportion: 800', props2)
results.hist(bins=np.arange(0.65, 0.85, 0.01))
```



两个分布都大致是正态，但一个比另一个更窄。 样本量为 800 的比例，比样本量为 200 的比例更紧密地聚集在 0.75 左右。增加样本量可以减少样本比例的可变性。

这应该不会令人惊讶。 我们多次产生了这样的直觉，更大的样本量通常会降低统计量的可变性。 然而，在样本均值的案例中，我们可以量化样本量和可变性之间的关系。

样本量究竟是如何影响样本均值或比例的可变性呢？ 这是我们将在下一节中讨论的问题。

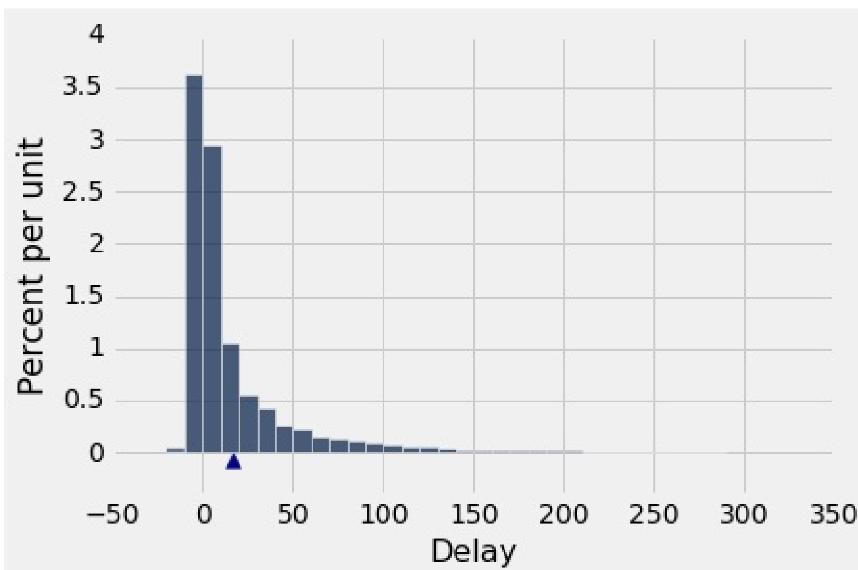
样本均值的可变性

根据中心极限定理，大型随机样本的均值的概率分布是大致正态的。 钟形曲线以总体平均值为中心。 一些样本均值较高，有些则较低，但距离总体均值的偏差在两边大致对称，正如我们已经看到的那样。 形式上，概率论表明样本均值是总体均值的无偏估计。

在我们的模拟中，我们也注意到较大样本的均值，相对较小样本的平均值更倾向于紧密聚集于总体均值附近。 在本节中，我们将量化样本均值的可变性，并建立可变性和样本量之间的关系。

我们从航班延误表开始。 平均延误时间约为 16.7 分钟，延误分布右倾。

```
united = Table.read_table('united_summer2015.csv')
delay = united.select('Delay')
pop_mean = np.mean(delay.column('Delay'))
pop_mean
16.658155515370705
```



现在我们来随机抽样，来查看样本均值的概率分布。 像往常一样，我们将使用模拟来得到这种分布的经验近似。

我们将定义一个函数 `simulate_sample_mean` 来实现，因为我们将在稍后改变样本量。 参数是表的名称，包含变量的列标签，样本量和模拟次数。

```
"""Empirical distribution of random sample means"""

def simulate_sample_mean(table, label, sample_size, repetitions):

    means = make_array()

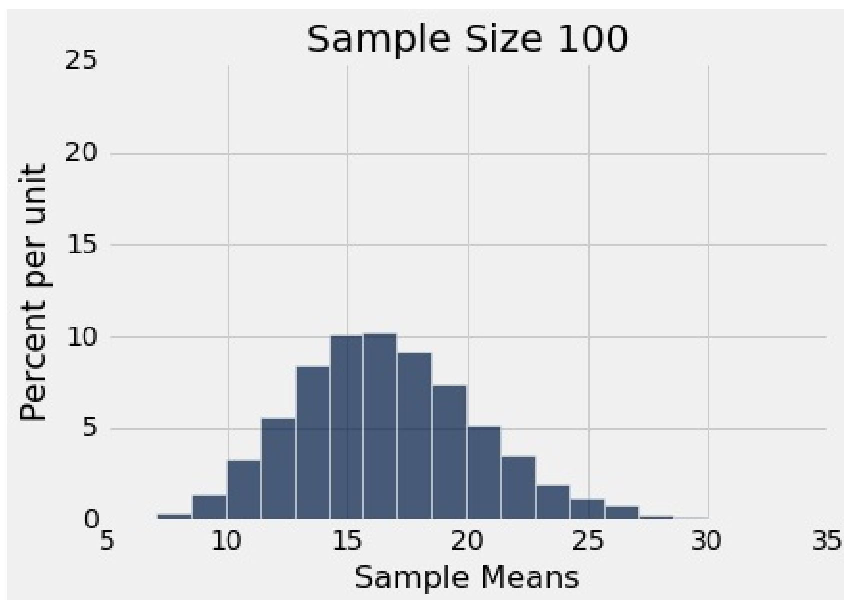
    for i in range(repetitions):
        new_sample = table.sample(sample_size)
        new_sample_mean = np.mean(new_sample.column(label))
        means = np.append(means, new_sample_mean)

    sample_means = Table().with_column('Sample Means', means)

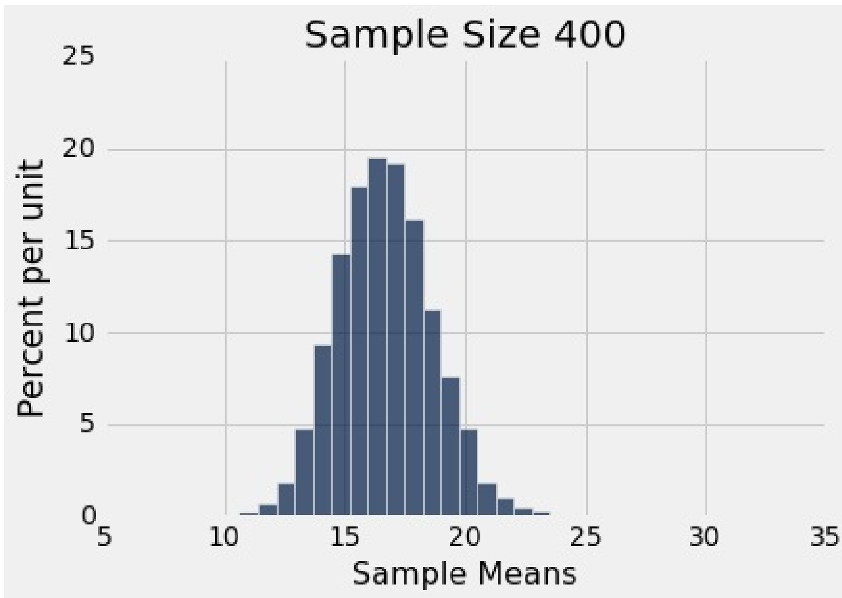
    # Display empirical histogram and print all relevant quantities
    sample_means.hist(bins=20)
    plots.xlabel('Sample Means')
    plots.title('Sample Size ' + str(sample_size))
    print("Sample size: ", sample_size)
    print("Population mean:", np.mean(table.column(label)))
    print("Average of sample means: ", np.mean(means))
    print("Population SD:", np.std(table.column(label)))
    print("SD of sample means:", np.std(means))
```

让我们模拟 100 个延误的随机样本的均值，然后是 400 个，最后是 625 个延误的均值。我们将对这些过程中的每一个执行 10,000 次重复。 `xlim` 和 `ylim` 在所有图表中设置一致的坐标轴，以便比较。你可以忽略每个单元格中的这两行代码。

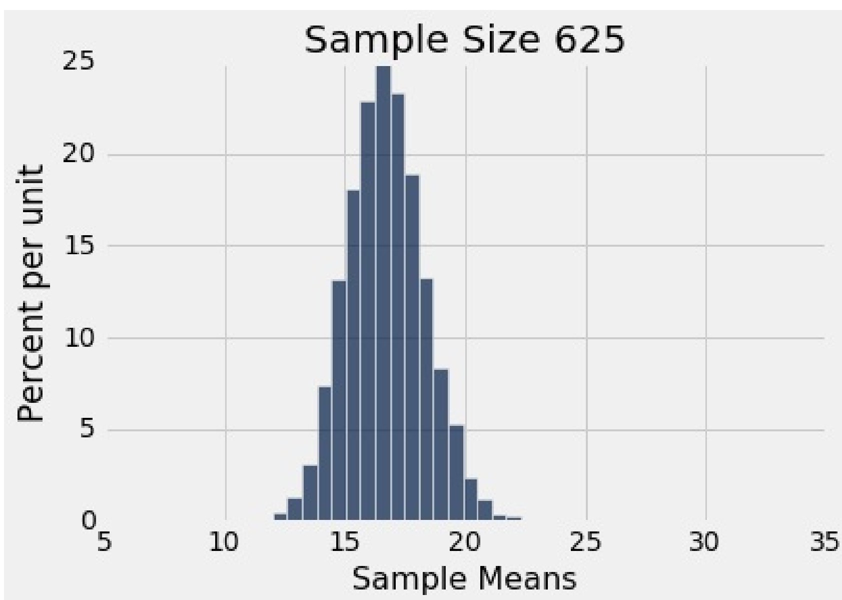
```
simulate_sample_mean(delay, 'Delay', 100, 10000)
plots.xlim(5, 35)
plots.ylim(0, 0.25);
Sample size:  100
Population mean: 16.6581555154
Average of sample means:  16.662059
Population SD: 39.4801998516
SD of sample means: 3.90507237968
```



```
simulate_sample_mean(delay, 'Delay', 400, 10000)
plots.xlim(5, 35)
plots.ylim(0, 0.25);
Sample size:  400
Population mean: 16.6581555154
Average of sample means:  16.67117625
Population SD: 39.4801998516
SD of sample means: 1.98326299651
```




```
simulate_sample_mean(delay, 'Delay', 625, 10000)
plots.xlim(5, 35)
plots.ylim(0, 0.25);
Sample size:  625
Population mean: 16.6581555154
Average of sample means:  16.68523712
Population SD: 39.4801998516
SD of sample means: 1.60089096006
```



你可以在实践中看到中心极限定律 - 样本均值的直方图是大致正态的，即使延误本身的直方图与正态分布相差甚远。

你还可以看到，样本均值的三个直方图中的每一个中心都非常接近总体均值。 在每种情况下，“样本均值的均值”非常接近 16.66 分钟，是总体均值。 每个直方图上方的打印输出都提供了这两个值。 像预期一样，样本均值是对总体均值的无偏估计。

所有样本均值的 SD

随着样本量的增加，你还可以看到直方图变窄，因此更高。 我们之前已经看到，但现在我们将更加关注延展度的度量。

所有延误总体的标准差约为 40 分钟。

```
pop_sd = np.std(delay.column('Delay'))
pop_sd
39.480199851609314
```

看看上面的样本均值的直方图中的标准差。在这三个里面，延误总体的标准差约为 40 分钟，因为所有的样本都来自同一个总体。

现在来看，样本量为 100 时，所有 10,000 个样本均值的标准差。标准差是总体标准差的十分之一。当样本量为 400 时，所有样本均值的标准差约为总体标准差的二十分之一。当样本量为 625 时，样本均值的标准差为总体标准差的二十五分之一。

将样本均值的经验分布的标准差与“总体标准差除以样本量的平方根”的数量进行比较，似乎是一个好主意。

这里是数值。对于第一列中的每个样本量，抽取 10,000 个该大小的随机样本，并计算 10,000 个样本均值。第二列包含那些 10,000 个样本均值的标准差。第三列包含计算结果“总体标准差除以样本量的平方根”。

该单元格需要一段时间来运行，因为这是大型模拟。但是你很快就会看到它值得等待。

```
repetitions = 10000
sample_sizes = np.arange(25, 626, 25)

sd_means = make_array()

for n in sample_sizes:
    means = make_array()
    for i in np.arange(repetitions):
        means = np.append(means, np.mean(delay.sample(n).column('Delay')))
    sd_means = np.append(sd_means, np.std(means))

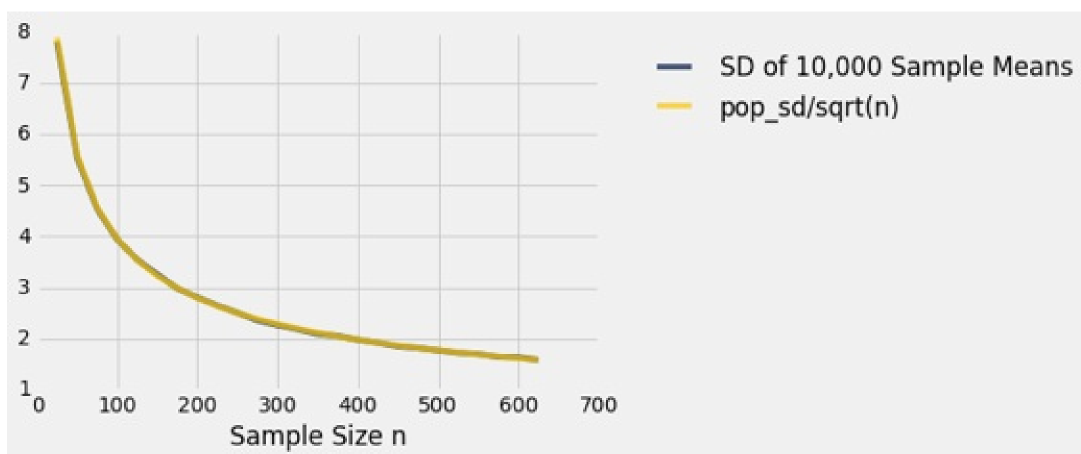
sd_comparison = Table().with_columns(
    'Sample Size n', sample_sizes,
    'SD of 10,000 Sample Means', sd_means,
    'pop_sd/sqrt(n)', pop_sd/np.sqrt(sample_sizes)
)
sd_comparison
```

Sample Size n	SD of 10,000 Sample Means	pop_sd/sqrt(n)
25	7.95017	7.89604
50	5.53425	5.58334
75	4.54429	4.55878
100	3.96157	3.94802
125	3.51095	3.53122
150	3.23949	3.22354
175	3.00694	2.98442
200	2.74606	2.79167
225	2.63865	2.63201
250	2.51853	2.49695

（省略了 15 行）

第二列和第三列的值非常接近。 如果我们用横轴上的样本量绘制每个列，那么这两个图基本上是不可区分的。

```
sd_comparison.plot('Sample Size n')
```



那里确实有两条曲线。但他们彼此如此接近，看起来好像只有一个。

我们看到了一个普遍结果的实例。 请记住，上面的图表基于每个样本量的 10,000 个重复。 但是每个样本量有超过 10,000 个样本。 样本均值的概率分布基于大小固定的所有可能样本的均值。

固定样本大小。如果样本是从总体中带放回随机抽取的：

$$\text{SD of all possible sample means} ~=~ \frac{\text{Population SD}}{\sqrt{\text{sample size}}}$$

这是所有可能样本均值的标准差。 它大致衡量了样本均值与总体均值的差距。

用于样本均值的中心极限定律

如果从总体中带放回地抽取大型随机样本，那么不管总体分布情况如何，样本均值的概率分布大致是正态的，以总体均值为中心，标准等于总体标准差除以样本量的平方根。

样本均值的准确性

所有可能的样本均值的标准差表示样本均值的变化程度。因此，它被视为样本均值作为总体均值的估计的准确度的一个度量。标准差越小，估计越准确。

公式表明：

- 总体大小不影响样本均值的准确性。公式中的任何地方都没有出现总体大小。
- 总体标准差是一个常数；从总体中抽取的每个样本都是一样的。样本量可以变化。由于样本量出现在分母中，样本均值的可变性随着样本量的增加而降低，因此准确度增加。

平方根法则

从标准差比较表中可以看出，25 次航班延误的随机样本的均值的标准差约为 8 分钟。 如果你将样本量乘以 4，你将得到大小为 100 的样本。所有这些样本的均值的标准差约为 4 分钟。这比 8 分钟还小，但并不是 4 倍，只有 2 倍。 这是因为分母中的样本量上面有一个平方根。样本量增加了 4 倍，但标准差下降了 `2 = sqrt(4)` 倍。 换句话说，准确度上升了 `2 = sqrt(4)` 倍。

一般来说，当你将样本量乘以一个因数时，样本均值的准确度将会上升该因数的平方根。

所以为了提高 10 倍的准确度，你必须将样本量乘以 100 倍。精度并不便宜！

选取样本量

候选人 A 在大选中竞选。一个投票机构想要估计投票给她的选民的比例。假设他们打算随机抽取选民，但实际上他们的抽样方法会更复杂。他们如何决定样本应该多大，才能达到理想的准确度？

在作出一些假设之后，我们现在可以回答这个问题：

- 选民人数非常多，所以我们可以假定随机样本带放回地抽取。
- 投票机构将通过为候选人 A 的选民百分比，构建一个约 95% 置信区间来做出估计。
- 准确度的理想水平是间隔宽度不应超过 1%。这非常准确！例如，置信区间 `(33.2%, 34%)` 可以，但 `(33.2%, 35%)` 不行。
- 我们将以候选人 A 的选民比例为例。回想一下，比例是一个平均值，其中总体中的值只

有 0（你不计算的个体类型）或 1（你计算的个体类型）。

置信区间的宽度

如果我们有一个随机样本，我们可以使用自举法为候选人 A 的选民百分比构建一个置信区间。但是我们还没有样本 - 我们试图找出样本有多大，为了让我们的置信区间如我们所希望的那样狭窄。

在这样的情况下，了解理论预测的结果会有帮助。

中心极限定律表明，样本比例大致是正态分布的，以总体中 1 的比例为中心，标准差等于总体中 0 和 1 的标准差除以样本量的平方根。

所以即使我们不能把自己的目标作为自举比例的第 2.5 和第 97.5 个百分点，那么置信区间仍然是正态分布的“中间 95%”。

有没有另外一种方法来求出间隔有多大？是的，因为我们知道对于正态分布变量，“中心上下两个标准差”的间隔包含 95% 的数据。

置信区间将延伸到样本比例的两个标准差，位于中心的任一侧。因此，间隔的宽度将是样本比例的 4 个标准差。

我们愿意容忍 `1% = 0.01` 的宽度。因此，使用上一节中开发的公式：

$$4 \times \frac{\text{SD of the 0-1 population}}{\sqrt{\text{sample size}}} ~\le~ 0.01)$$

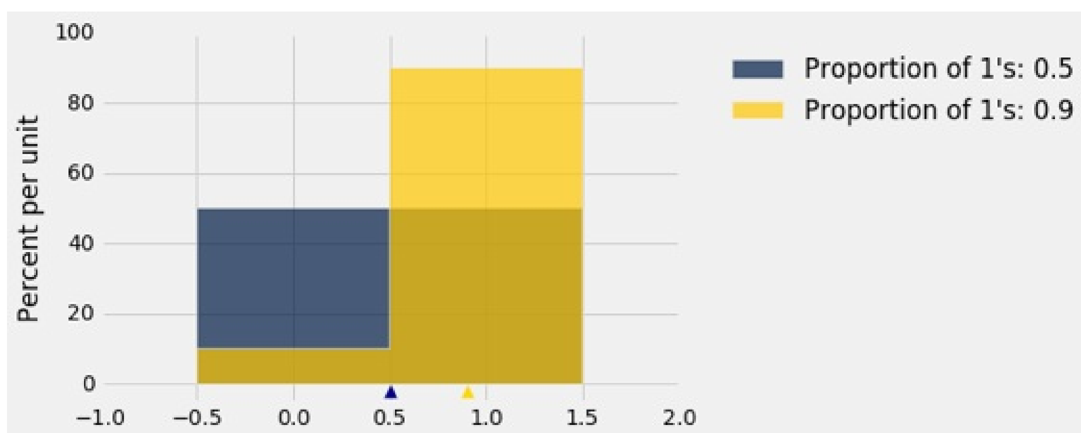
所以：

$$\sqrt{\text{sample size}} ~\ge~ 4 \times \frac{\text{SD of the 0-1 population}}{0.01}$$

01 集合的标准差

如果我们知道总体的标准差，我们就完成了。 我们可以计算样本量的平方根，然后取平方得到样本量。 但我们不知道总体的标准差。 总体中，候选人 A 的每个选民为 1，其余选民为 0，我们不知道每种选民的比例是多少。 这就是我们正在估计的。

那么我们卡住了吗？ 不，因为我们可以限制人口的标准差。 这里是两个这样的分布的直方图，一个是相等比例的 1 和 0 ，另一个是 90% 的 1 和 10% 的 0。 哪一个标准差更大？



请记住，总体中的可能值只有 0 和 1。

蓝色直方图（50% 的 1 和 50% 的 0）比金色延展度更大。 它的均值是 0.5。 距离均值的偏差，一半等于 0.5，另一半等于 -0.5，所以标准差是 0.5。

在金色直方图中，所有的区域都挤压在 1 左右，从而延展度更小。 90% 的偏差很小，为 0.1。 其他的 10% 是 -0.9 ，较大，但总体上的延展度比蓝色直方图小。

如果我们改变 1 的比例或者让 0 的比例大于 1 的比例，那么同样的观察也成立。 我们通过计算不同比例，只包含 0 和 1 的 10 个元素的总体的标准差来检查它。 函数 `np.ones` 对此很有用。 它接受一个正整数作为它的参数，并返回一个由多个 1 组成的数组。

```
sd = make_array()
for i in np.arange(1, 10, 1):
    # Create an array of i 1's and (10-i) 0's
    population = np.append(np.ones(i), 1-np.ones(10-i))
    sd = np.append(sd, np.std(population))

zero_one_sds = Table().with_columns(
    "Population Proportion of 1's", np.arange(0.1, 1, 0.1),
    "Population SD", sd
)

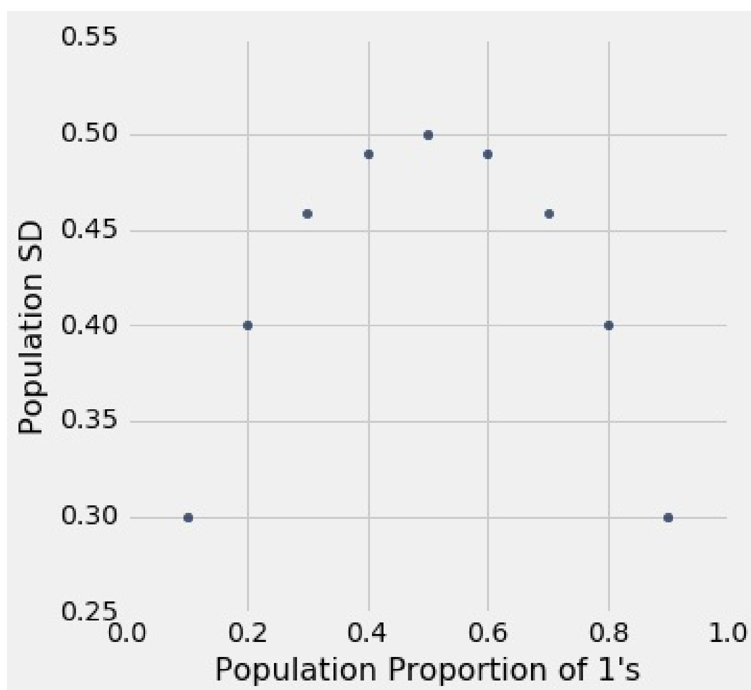
zero_one_sds
```

Population Proportion of 1's	Population SD
0.1	0.3
0.2	0.4
0.3	0.458258
0.4	0.489898
0.5	0.5
0.6	0.489898
0.7	0.458258
0.8	0.4
0.9	0.3

毫不奇怪，10% 的 1 和 90% 的 0 的总体标准差，与 90% 的 1 和 10% 的 0 的总体标准差相同。 那是因为你把直方图的一个条和两一个条互换，延展度没有变化。

更重要的是，出于我们的目的，标准差随着 1 的比例增加而增加，直到 1 的比例为 0.5；然后开始对称下降。

```
zero_one_sds.scatter("Population Proportion of 1's")
```



总结：01 总体的标准差最大为 0.5。 当 50% 的总体为 1 而另外 50% 为 0 时，这就是标准差的值。

样本量

我们知道了 $\sqrt{\mbox{sample size}} ~\ge~ 4 \times \frac{\mbox{SD of the 0-1 population}}{0.01}$，并且 01 总体的标准差最大为 0.5，无论总体中 1 的比例。所以这样是安全的：

$$\sqrt{\mbox{sample size}} ~\ge~ 4 \times \frac{0.5}{0.01} ~=~ 200$$

所以样本量应该至少是`200 ^ 2 = 40,000`。这是一个巨大的样本！但是，如果你想以较高的置信度确保高精度，不管总体是什么样子，那就是你所需要的。

十三、预测

> 原文：Prediction
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

数据科学的一个重要方面，是发现数据可以告诉我们什么未来的事情。气候和污染的数据说了几十年内温度的什么事情？根据一个人的互联网个人信息，哪些网站可能会让他感兴趣？病人的病史如何用来判断他或她对治疗的反应？

为了回答这样的问题，数据科学家已经开发出了预测的方法。在本章中，我们将研究一种最常用的方法，基于一个变量的值来预测另一个变量。

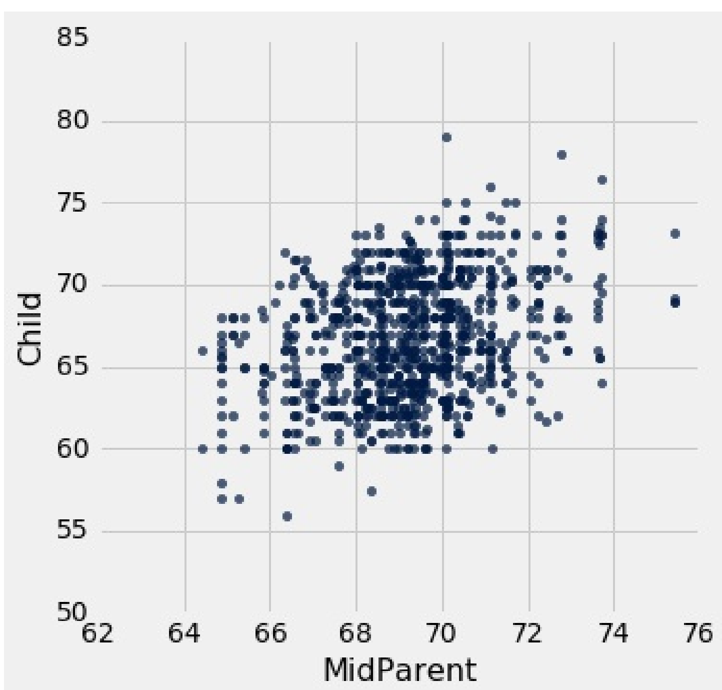
方法的基础由弗朗西斯·高尔顿爵士（Sir Francis Galton）奠定。我们在 7.1 节看到，高尔顿研究了身体特征是如何从一代传到下一代的。他最著名的工作之一，是根据父母的高度预测子女的身高。我们已经研究了高尔顿为此收集的数据集。`heights` 表包含了 934 个成年子女的双亲身高和子女身高（全部以英寸为单位）。

```
# Galton's data on heights of parents and their adult children
galton = Table.read_table('galton.csv')
heights = Table().with_columns(
    'MidParent', galton.column('midparentHeight'),
    'Child', galton.column('childHeight')
    )
heights
```

MidParent	Child
75.43	73.2
75.43	69.2
75.43	69
75.43	69
73.66	73.5
73.66	72.5
73.66	65.5
73.66	65.5
72.06	71
72.06	68

（省略了 924 行）

```
heights.scatter('MidParent')
```



收集数据的主要原因是能够预测成年子女的身高，他们的父母与数据集中相似。 在注意到两个变量之间的正相关之后，我们在第 7.1 节中做了这些预测。

我们的方法是，基于新人的双亲身高周围的所有点来做预测。 为此，我们编写了一个名为 `predict_child` 的函数，该函数以双亲身高作为参数，并返回双亲身高在半英寸之内的，所有子女的平均身高。

```
def predict_child(mpht):
    """Return a prediction of the height of a child
    whose parents have a midparent height of mpht.

    The prediction is the average height of the children
    whose midparent height is in the range mpht plus or minus 0.5 inches.
    """

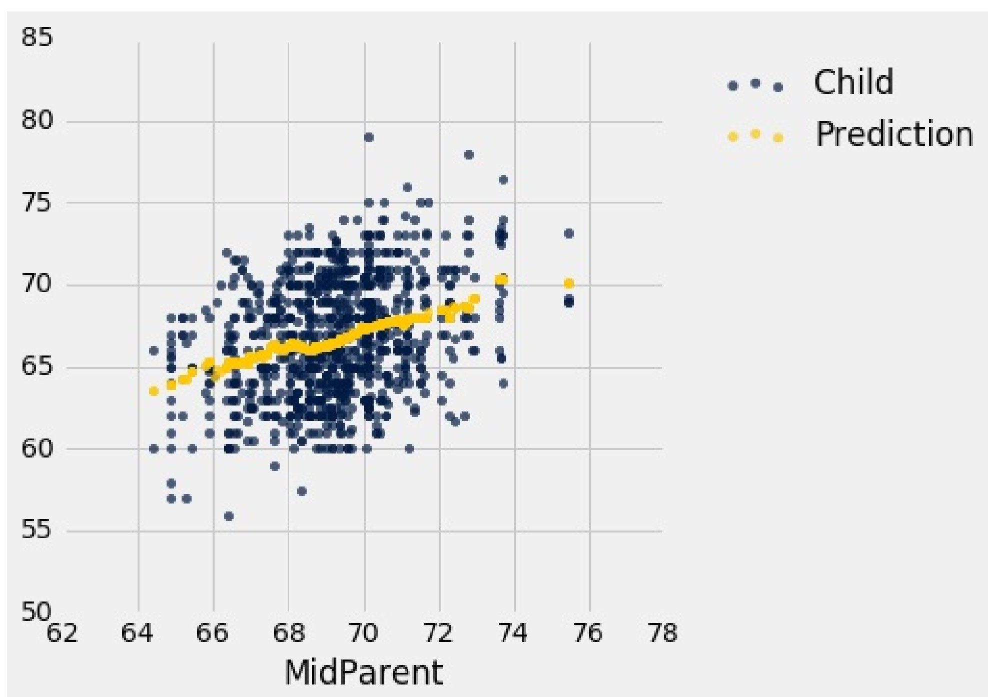
    close_points = heights.where('MidParent', are.between(mpht-0.5, mpht + 0.5))
    return close_points.column('Child').mean()
```

我们将函数应用于 `Midparent` 列，可视化我们的结果。

```
# Apply predict_child to all the midparent heights

heights_with_predictions = heights.with_column(
    'Prediction', heights.apply(predict_child, 'MidParent')
    )
# Draw the original scatter plot along with the predicted values

heights_with_predictions.scatter('MidParent')
```



给定双亲身高的预测值，大致位于给定身高处的垂直条形的中心。这种预测方法称为回归。本章后面我们会看到这个术语的来源。 我们也会看到，我们是否可以避免将“接近”任意定义为“在半英寸之内”。 但是首先我们要开发一个可用于很多环境的方法，来决定一个变量作为另一个变量的预测值有多好。

相关性

在本节中，我们将开发一种度量，度量散点图紧密聚集在一条直线上的程度。 形式上，这被称为测量线性关联。

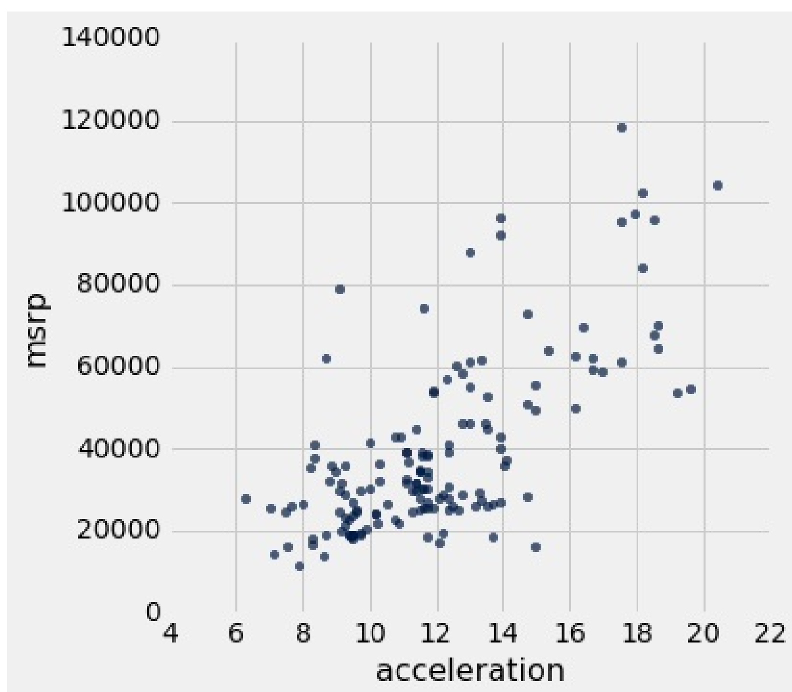
`hybrid` 表包含了 1997 年到 2013 年在美国销售的混合动力车的数据。数据来自佛罗里达大学 Larry Winner 教授的在线数据档案。这些列为：

- `vehicle` ：车的型号
- `year` ：出厂年份
- `msrp`：2013 年制造商的建议零售价（美元）
- `acceleration`：加速度（千米每小时每秒）
- `mpg`：燃油效率（英里每加仑）
- `class`：型号的类别

（省略了 143 行）

下图是 `msrp` 与 `acceleration` 的散点图。 这意味着 `msrp` 绘制在纵轴上并且 `acceleration` 在横轴上。

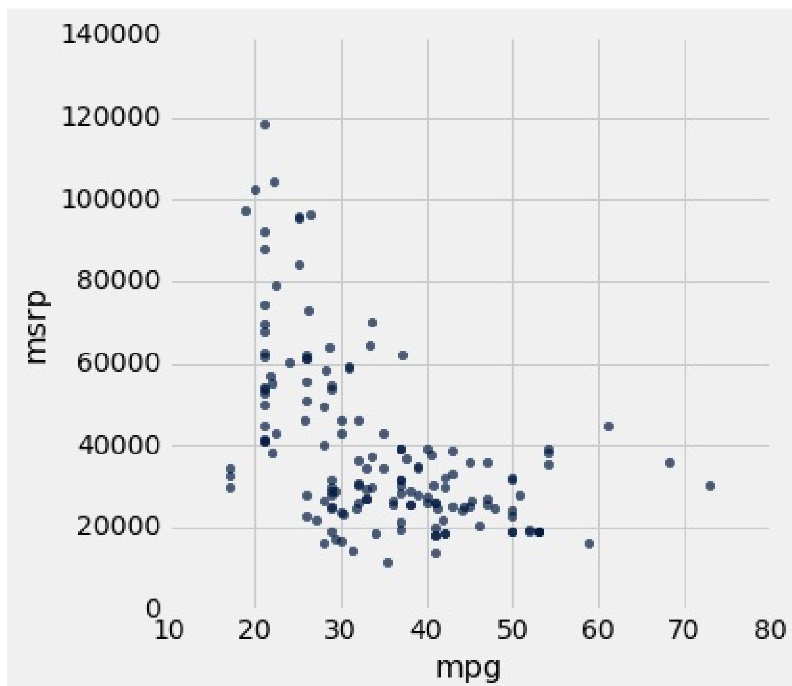
```
hybrid.scatter('acceleration', 'msrp')
```



注意正相关。 散点图倾斜向上，表明加速度较大的车辆通常成本更高；相反，价格更高的汽车通常具有更大的加速。

`msrp` 与 `mpg` 的散点图表明了负相关。 `mpg` 较高的混合动力车往往成本较低。 这似乎令人惊讶，直到你明白了，加速更快的汽车往往燃油效率更低，行驶里程更低。 之前的散点图显示，这些也是价格更高的车型。

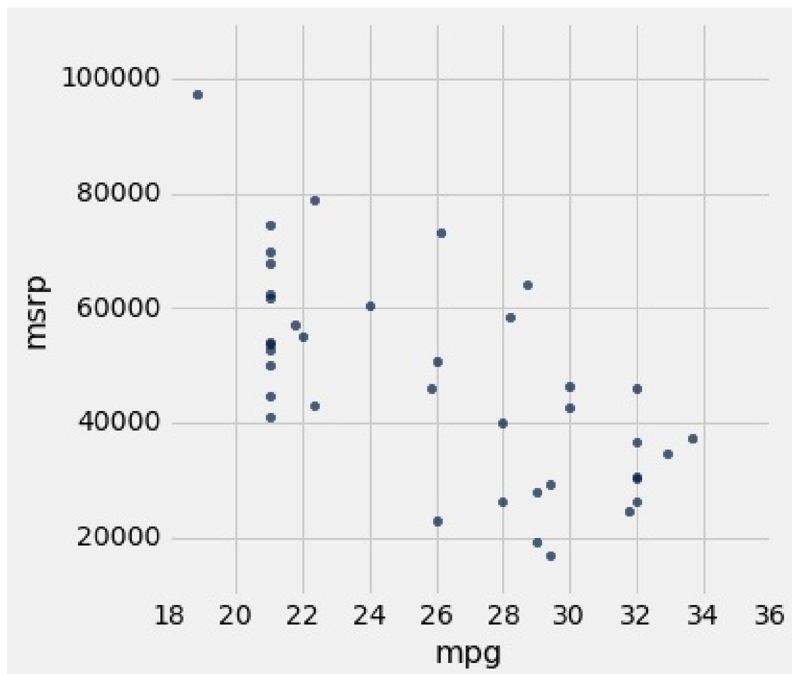
```
hybrid.scatter('mpg', 'msrp')
```



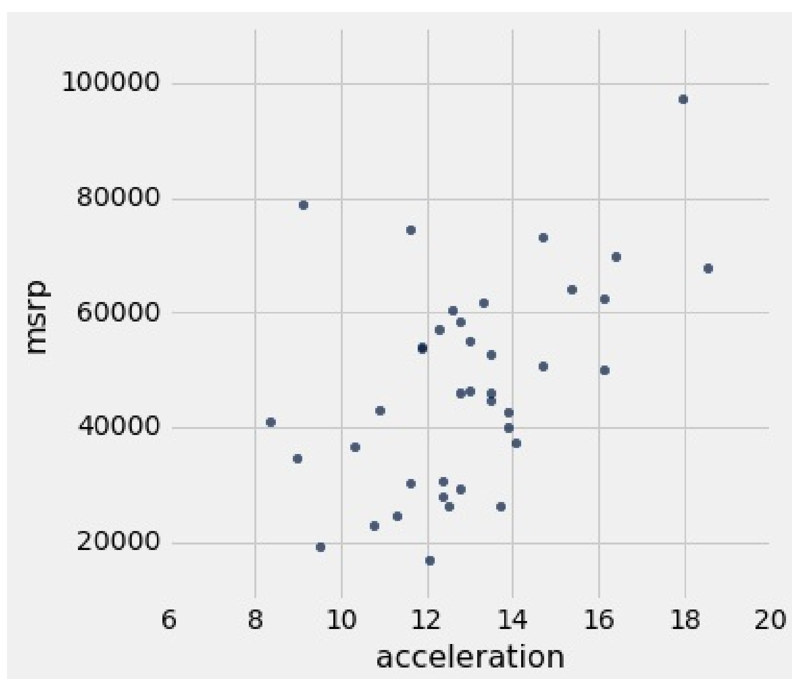
除了负相关，价格与效率的散点图显示了两个变量之间的非线性关系。 这些点似乎围绕在一条曲线周围，而不是一条直线。

但是，如果我们只将数据限制在 SUV 类别中，价格和效率之间仍然负相关的，但是这种关系似乎更为线性。 SUV 价格与加速度之间的关系也呈线性趋势，但是斜率是正的。

```
suv = hybrid.where('class', 'SUV')
suv.scatter('mpg', 'msrp')
```




```
suv.scatter('acceleration', 'msrp')
```



你会注意到，即使不关注变量被测量的单位，我们也可以从散点图的大体方向和形状中得到有用的信息。

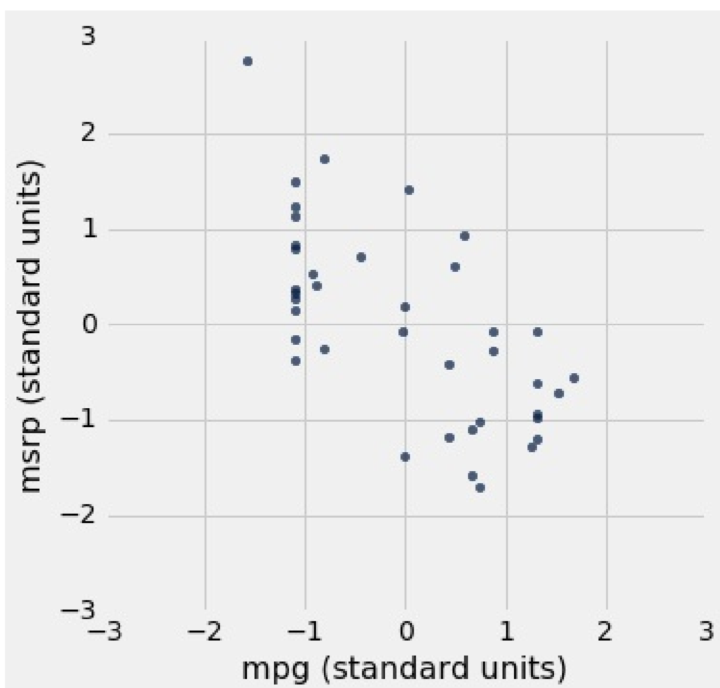
事实上，我们可以将所有的变量绘制成标准单位，并且绘图看起来是一样的。 这给了我们一个方法，来比较两个散点图中的线性程度。

回想一下，在前面的章节中，我们定义了`standard_units`函数来将数值数组转换为标准单位。

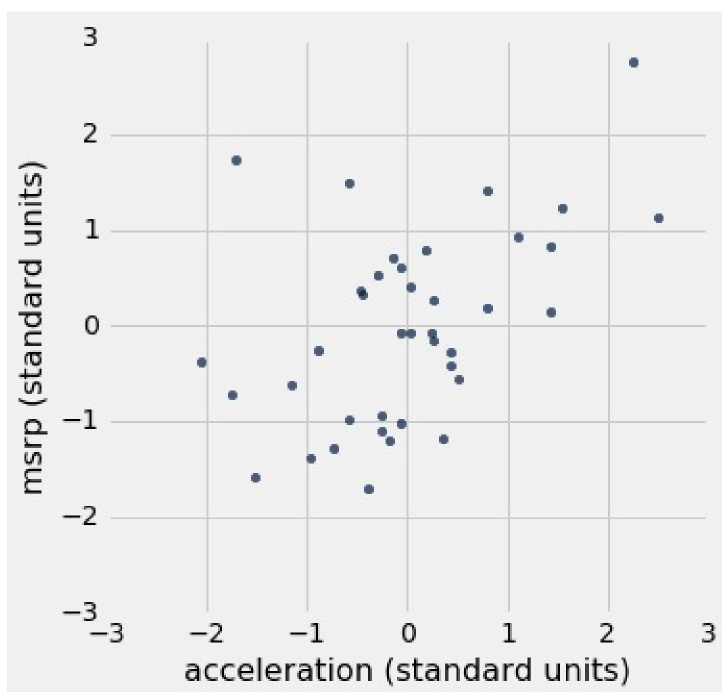
```
def standard_units(any_numbers):
    "Convert any array of numbers to standard units."
    return (any_numbers - np.mean(any_numbers))/np.std(any_numbers)
```

我们可以使用这个函数重新绘制 SUV 的两个散点图，所有变量都以标准单位测量。

```
Table().with_columns(
    'mpg (standard units)',  standard_units(suv.column('mpg')),
    'msrp (standard units)', standard_units(suv.column('msrp'))
).scatter(0, 1)
plots.xlim(-3, 3)
plots.ylim(-3, 3);
```




```
Table().with_columns(
    'acceleration (standard units)', standard_units(suv.column('acceleration')),
    'msrp (standard units)',         standard_units(suv.column('msrp'))
).scatter(0, 1)
plots.xlim(-3, 3)
plots.ylim(-3, 3);
```



我们在这些数字中看到的关联与我们之前看到的一样。 另外，由于现在两张散点图的刻度完全相同，我们可以看到，第二张图中的线性关系比第一张图中的线性关系更加模糊。

我们现在将定义一个度量，使用标准单位来量化我们看到的这种关联。

相关系数

相关系数测量两个变量之间线性关系的强度。 在图形上，它测量散点图聚集在一条直线上的程度。

相关系数这个术语不容易表述，所以它通常缩写为相关性并用 `r` 表示。

以下是一些关于 `r` 的数学事实，我们将通过模拟观察。

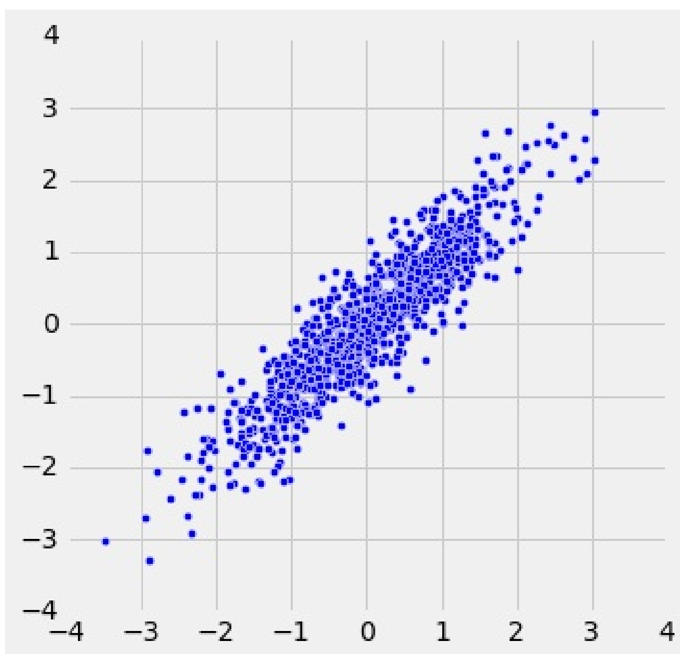
- 相关系数 `r` 是介于 `-1` 和 `1` 之间的数字。
- `r` 度量了散点图围绕一条直线聚集的程度。
- 如果散点图是完美的向上倾斜的直线，`r = 1`，如果散点图是完美的向下倾斜的直线，`r = -1`。

函数 `r_scatter` 接受 `r` 值作为参数，模拟相关性非常接近 `r` 的散点图。 由于模拟中的随机性，相关性不会完全等于 `r`。

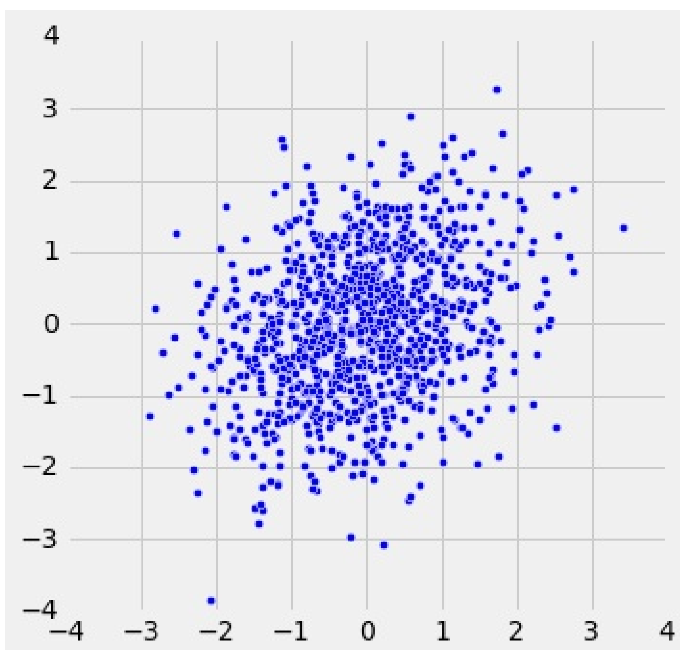
调用 `r_scatter` 几次，以 `r` 的不同值作为参数，并查看散点图如何变化。

当 `r = 1` 时，散点图是完全线性的，向上倾斜。 当 `r = -1` 时，散点图是完全线性的，向下倾斜。 当 `r = 0` 时，散点图是围绕水平轴的不定形云，并且变量据说是不相关的。

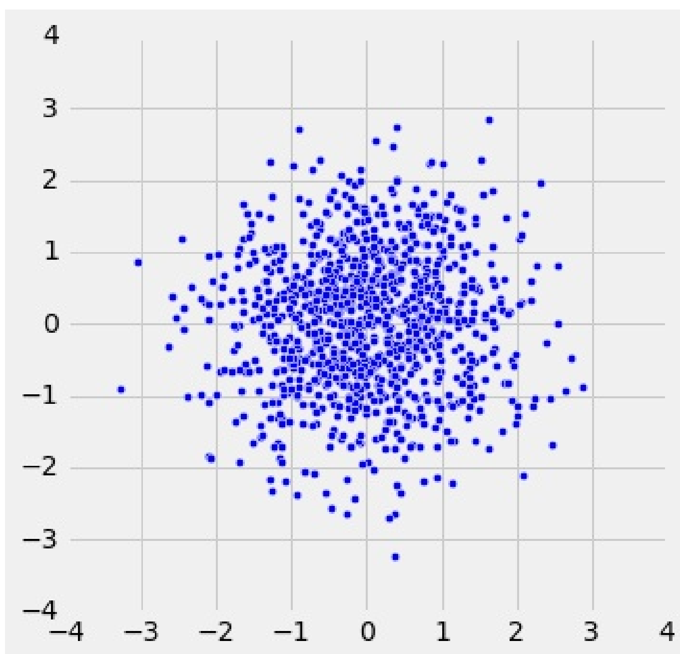
```
r_scatter(0.9)
```



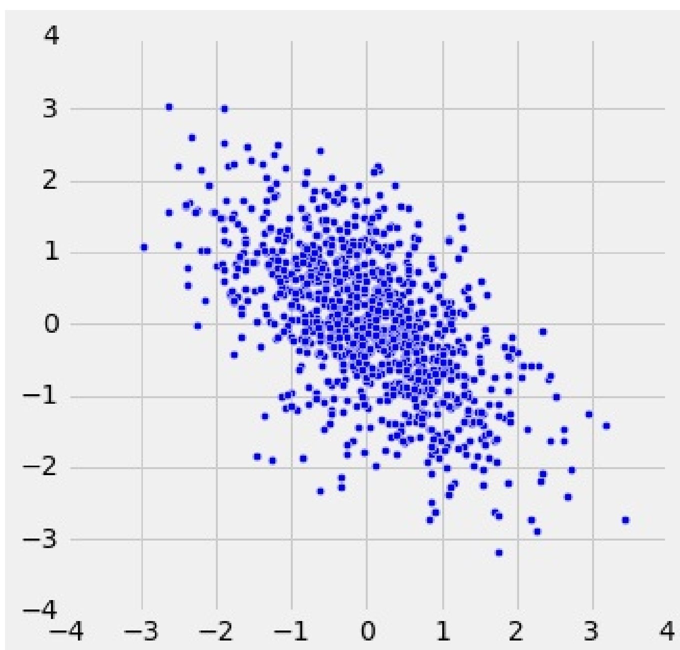

```
r_scatter(0.25)
```




```
r_scatter(0)
```




```
r_scatter(-0.55)
```



计算 `r`

目前为止，`r` 的公式还不清楚。 它拥有超出本课程范围的数学基础。 然而，你将会看到，这个计算很简单，可以帮助我们理解 `r` 的几个属性。

`r` 的公式：

`r` 是两个变量的乘积的均值，这两个变量都以标准单位来衡量。

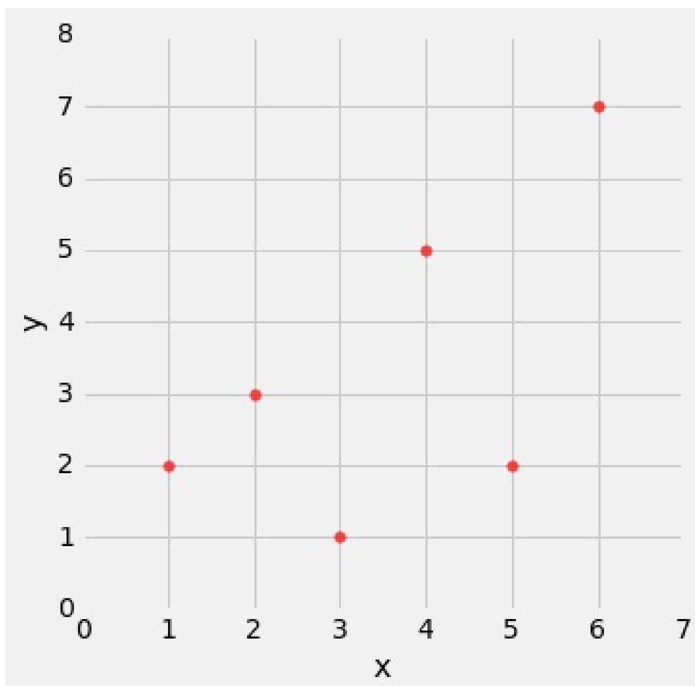
以下是计算中的步骤。 我们将把这些步骤应用于 `x` 和 `y` 值的简单表格。

```
x = np.arange(1, 7, 1)
y = make_array(2, 3, 1, 5, 2, 7)
t = Table().with_columns(
        'x', x,
        'y', y
    )
t
```

x	y
1	2
2	3
3	1
4	5
5	2
6	7

根据散点图，我们预计 `r` 将是正值，但不等于 1。

```
t.scatter(0, 1, s=30, color='red')
```



第一步：将每个变量转换为标准单位。

```
t_su = t.with_columns(
        'x (standard units)', standard_units(x),
        'y (standard units)', standard_units(y)
    )
t_su
```

x	y	x (standard units)	y (standard units)
1	2	-1.46385	-0.648886
2	3	-0.87831	-0.162221
3	1	-0.29277	-1.13555
4	5	0.29277	0.811107
5	2	0.87831	-0.648886
6	7	1.46385	1.78444

第二步：将每一对标准单位相乘

```
t_product = t_su.with_column('product of standard units', t_su.column(2) * t_su.column(
3))
t_product
```

x	y	x (standard units)	y (standard units)	product of standard units
1	2	-1.46385	-0.648886	0.949871
2	3	-0.87831	-0.162221	0.142481
3	1	-0.29277	-1.13555	0.332455
4	5	0.29277	0.811107	0.237468
5	2	0.87831	-0.648886	-0.569923
6	7	1.46385	1.78444	2.61215

第三步：`r` 是第二步计算的乘积的均值。

```
# r is the average of the products of standard units

r = np.mean(t_product.column(4))
r
0.61741639718977093
```

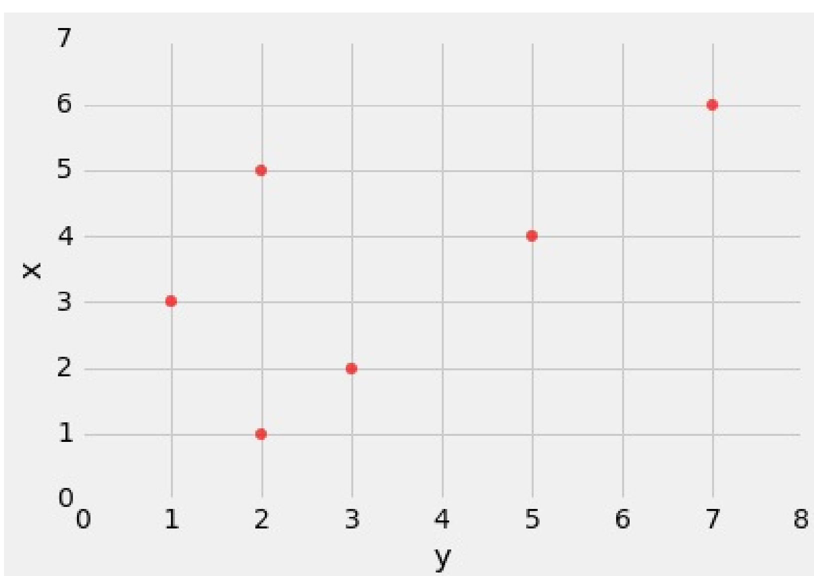
正如我们的预期，`r` 是个不等于的正值。

`r` 的性质

计算结果表明：

`r` 是一个纯数字。它没有单位。这是因为 `r` 基于标准单位。`r` 不受任何轴上单位的影响。这也是因为 `r` 基于标准单位。`r` 不受轴的交换的影响。在代数上，这是因为标准单位的乘积不依赖于哪个变量被称为 `x` 和 `y`。在几何上，轴的切换关于 `y = x` 直线翻转了散点图，但不会改变群聚度和关联的符号。

```
t.scatter('y', 'x', s=30, color='red')
```



`correlation` 函数

我们将要重复计算相关性，所以定义一个函数会有帮助，这个函数通过执行上述所有步骤来计算它。 让我们定义一个函数 `correlation` ，它接受一个表格，和两列的标签。该函数返回 `r` ，它是标准单位下这些列的值的乘积的平均值。

```
def correlation(t, x, y):
    return np.mean(standard_units(t.column(x))*standard_units(t.column(y)))
```

让我们在 `t` 的 `x` 和 `y` 列上调用函数。 该函数返回 `x` 和 `y` 之间的相关性的相同答案，就像直接应用 `r` 的公式一样。

```
correlation(t, 'x', 'y')
0.61741639718977093
```

我们注意到，变量被指定的顺序并不重要。

```
correlation(t, 'y', 'x')
0.61741639718977093
```

在 `suv` 表的列上调用 `correlation` ，可以使我们看到价格和效率之间的相关性，以及价格和加速度之间的相关性。

```
correlation(suv, 'mpg', 'msrp')
-0.6667143635709919
correlation(suv, 'acceleration', 'msrp')
0.48699799279959155
```

这些数值证实了我们的观察：

价格和效率之间存在负相关关系，而价格和加速度之间存在正相关关系。价格和加速度之间的线性关系（相关性约为 0.5），比价格和效率之间的线性关系稍弱（相关性约为 -0.67）。相关性是一个简单而强大的概念，但有时会被误用。在使用 `r` 之前，重要的是要知道相关性能做和不能做什么。

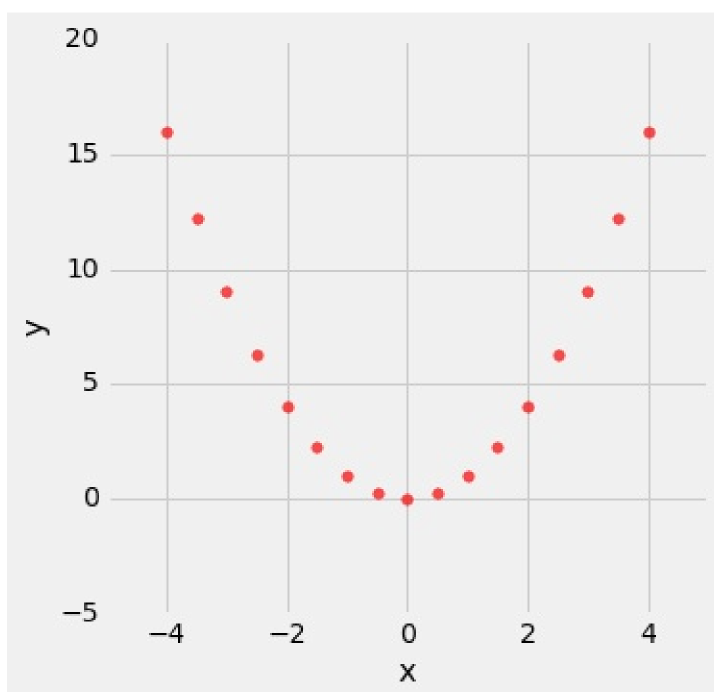
相关不是因果

相关只衡量关联，并不意味着因果。尽管学区内的孩子的体重与数学能力之间的相关性可能是正的，但这并不意味着做数学会使孩子更重，或者说增加体重会提高孩子的数学能力。年龄是一个使人混淆的变量：平均来说，较大的孩子比较小的孩子更重，数学能力更好。

相关性度量线性关联

相关性只测量一种关联 - 线性关联。具有较强非线性关联的变量可能具有非常低的相关性。这里有一个变量的例子，它具有完美的二次关联 `y = x ^ 2`，但是相关性等于 0。

```
new_x = np.arange(-4, 4.1, 0.5)
nonlinear = Table().with_columns(
        'x', new_x,
        'y', new_x**2
    )
nonlinear.scatter('x', 'y', s=30, color='r')
```

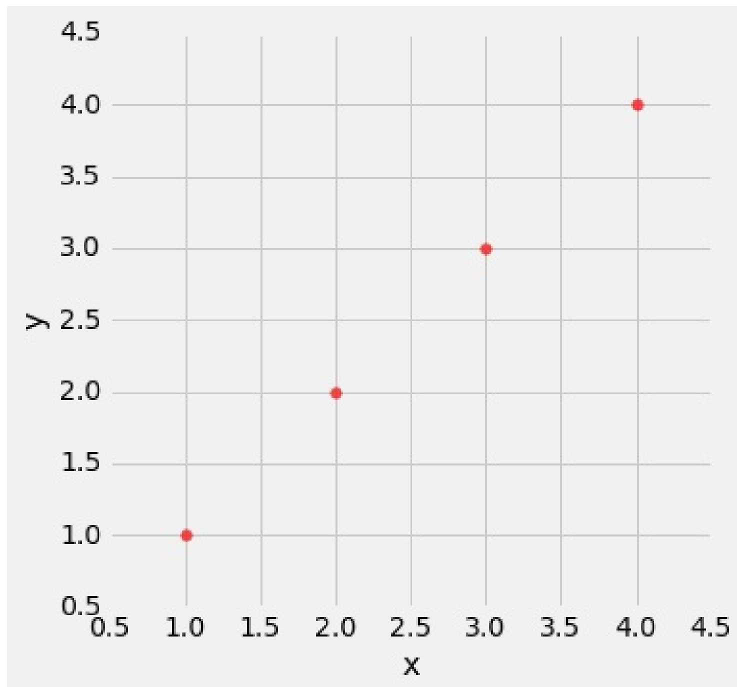



```
correlation(nonlinear, 'x', 'y')
0.0
```

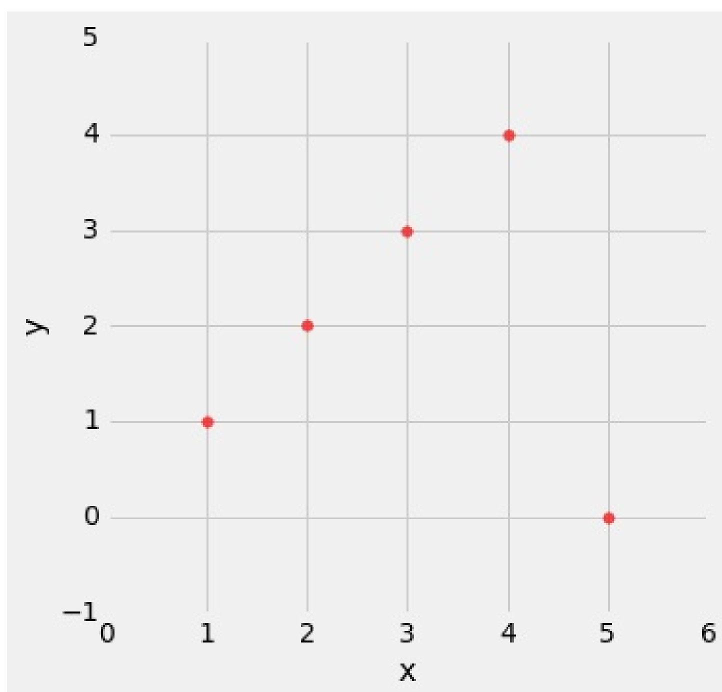
相关性受到离群点影响

离群点可能对相关性有很大的影响。 下面是一个例子，其中通过增加一个离群点， `r` 等于 1 的散点图变成 `r` 等于 0 的图。

```
line = Table().with_columns(
        'x', make_array(1, 2, 3, 4),
        'y', make_array(1, 2, 3, 4)
    )
line.scatter('x', 'y', s=30, color='r')
```




```
correlation(line, 'x', 'y')
1.0
outlier = Table().with_columns(
        'x', make_array(1, 2, 3, 4, 5),
        'y', make_array(1, 2, 3, 4, 0)
    )
outlier.scatter('x', 'y', s=30, color='r')
```




```
correlation(outlier, 'x', 'y')
0.0
```

生态相关性应谨慎解读

基于汇总数据的相关性可能会产生误导。 作为一个例子，这里是 2014 年 SAT 批判性阅读和数学成绩的数据。50 个州和华盛顿特区各有一个点。 `Participation Rate` 列包含参加考试的高中学生的百分比。 接下来的三列显示了每个州的测试每个部分的平均得分，最后一列是测试总得分的平均值。

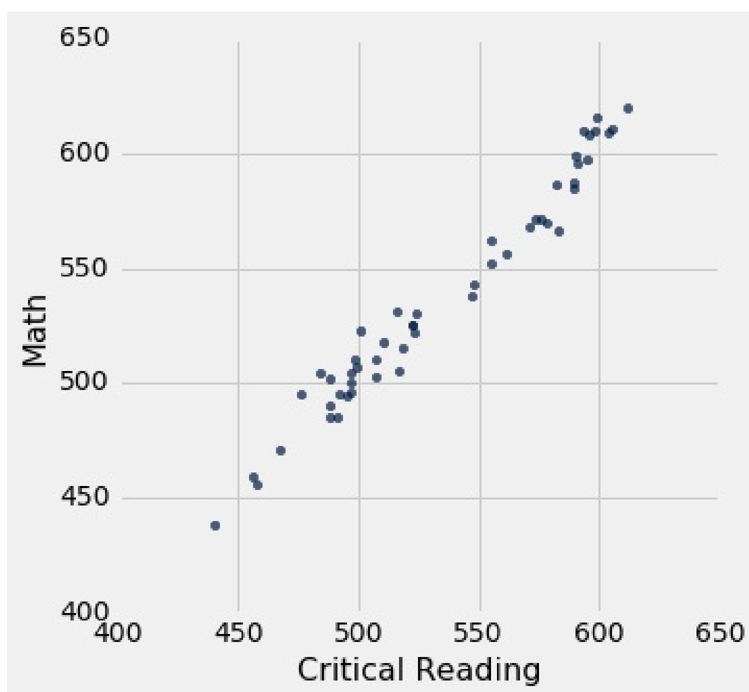
```
sat2014 = Table.read_table('sat2014.csv').sort('State')
sat2014
```

State	Participation Rate	Critical Reading	Math	Writing	Combined
Alabama	6.7	547	538	532	1617
Alaska	54.2	507	503	475	1485
Arizona	36.4	522	525	500	1547
Arkansas	4.2	573	571	554	1698
California	60.3	498	510	496	1504
Colorado	14.3	582	586	567	1735
Connecticut	88.4	507	510	508	1525
Delaware	100	456	459	444	1359
District of Columbia	100	440	438	431	1309
Florida	72.2	491	485	472	1448

（省略了 41 行）

数学得分与批判性阅读得分的散点图紧密聚集在一条直线上;相关性接近 0.985。

```
sat2014.scatter('Critical Reading', 'Math')
```




```
correlation(sat2014, 'Critical Reading', 'Math')
0.98475584110674341
```

这是个非常高的相关性。但重要的是要注意，这并不能反映学生的数学和批判性阅读得分之间的关系强度。

数据由每个州的平均分数组成。但是各州不参加考试 - 而是学生。表中的数据通过将每个州的所有学生聚集为（这个州里面的两个变量的均值处的）单个点而创建。但并不是所有州的学生都会在这个位置，因为学生的表现各不相同。如果你为每个学生绘制一个点，而不是每个州一个点，那么在上图中的每个点周围都会有一圈云状的点。整体画面会更模糊。学生的数学和批判性阅读得分之间的相关性，将低于基于州均值计算的数值。

基于聚合和均值的相关性被称为生态相关性，并且经常用于报告。正如我们刚刚所看到的，他们必须谨慎解读。

严重还是开玩笑？

2012 年，在著名的《新英格兰医学杂志》（New England Journal of Medicine）上发表的一篇论文，研究了一组国家巧克力消费与的诺贝尔奖之间的关系。《科学美国人》（Scientific American）严肃地做出回应，而其他人更加轻松。 欢迎你自行决定！下面的图表应该让你有兴趣去看看。

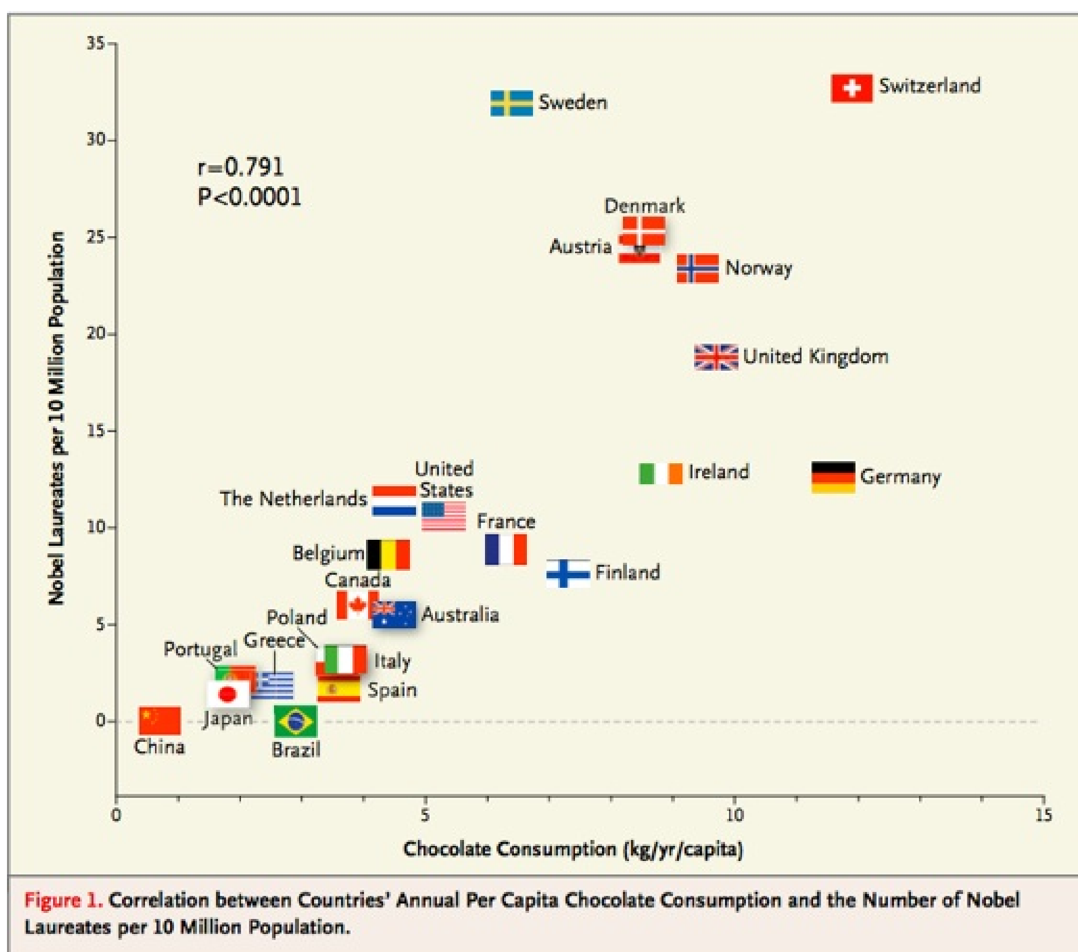


Figure 1. Correlation between Countries' Annual Per Capita Chocolate Consumption and the Number of Nobel Laureates per 10 Million Population.

回归直线

相关系数 `r` 并不只是测量散点图中的点聚集在一条直线上的程度。 它也有助于确定点聚集的直线。 在这一节中，我们将追溯高尔顿和皮尔逊发现这条直线的路线。

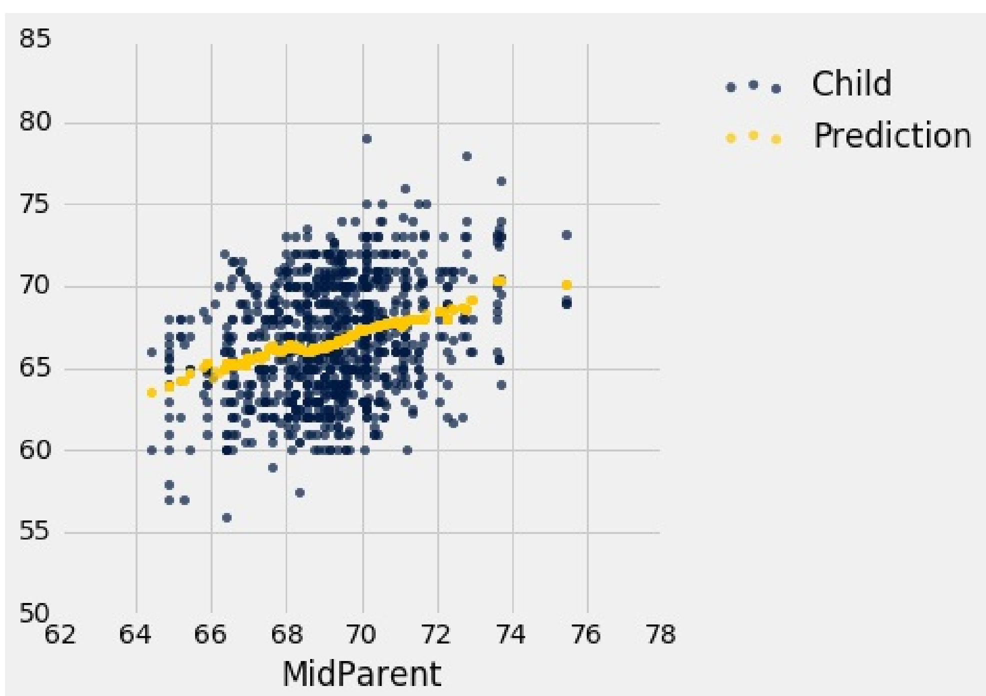
高尔顿的父母及其成年子女身高的数据显示出线性关系。 当我们基于双亲身高的子女身高的预测大致沿着直线时，就证实了线性。

```
galton = Table.read_table('galton.csv')

heights = Table().with_columns(
    'MidParent', galton.column('midparentHeight'),
    'Child', galton.column('childHeight')
    )
def predict_child(mpht):
    """Return a prediction of the height of a child
    whose parents have a midparent height of mpht.

    The prediction is the average height of the children
    whose midparent height is in the range mpht plus or minus 0.5 inches.
    """

    close_points = heights.where('MidParent', are.between(mpht-0.5, mpht + 0.5))
    return close_points.column('Child').mean()
heights_with_predictions = heights.with_column(
    'Prediction', heights.apply(predict_child, 'MidParent')
    )
heights_with_predictions.scatter('MidParent')
```



标准单位下的度量

让我们看看，我们是否能找到一个方法来确定这条线。 首先，注意到线性关联不依赖于度量单位 - 我们也可以用标准单位来衡量这两个变量。

```
def standard_units(xyz):
    "Convert any array of numbers to standard units."
    return (xyz - np.mean(xyz))/np.std(xyz)
heights_SU = Table().with_columns(
    'MidParent SU', standard_units(heights.column('MidParent')),
    'Child SU', standard_units(heights.column('Child'))
)
heights_SU
```

MidParent SU	Child SU
3.45465	1.80416
3.45465	0.686005
3.45465	0.630097
3.45465	0.630097
2.47209	1.88802
2.47209	1.60848
2.47209	-0.348285
2.47209	-0.348285
1.58389	1.18917
1.58389	0.350559

（省略了 924 行）

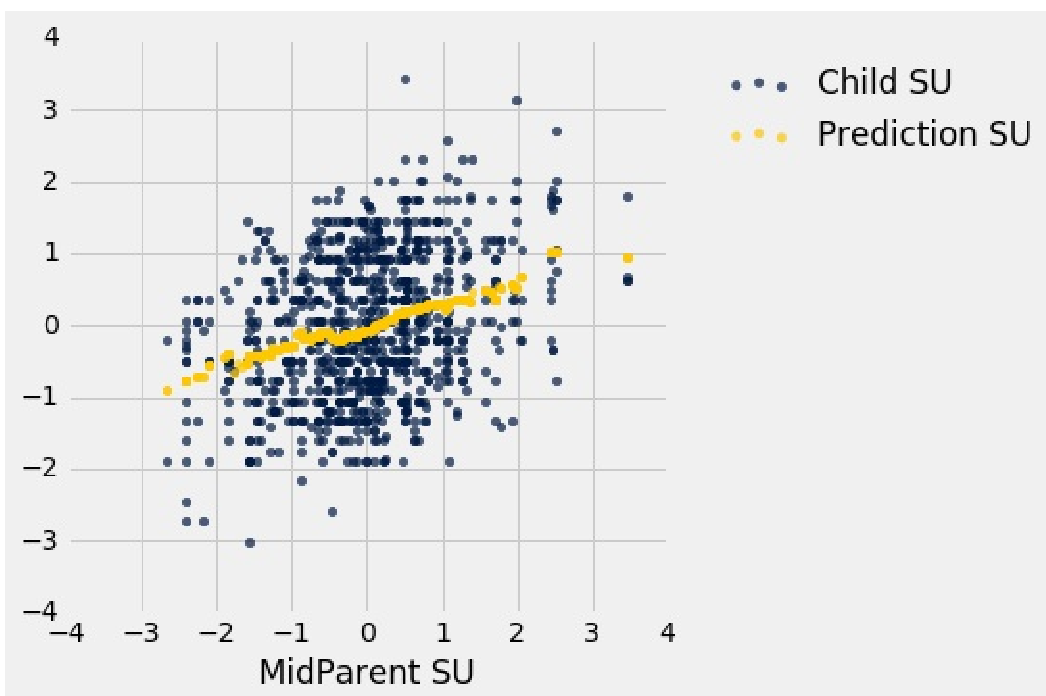
在这个刻度上，我们可以像以前一样精确地计算我们的预测。 但是首先我们必须弄清楚，如何将“接近”的点的旧定义转换为新的刻度上的一个值。 我们曾经说过，如果双亲高度在 0.5 英寸之内，它们就是“接近”的。 由于标准单位以标准差为单位测量距离，所以我们必须计算出，0.5 英寸是多少个双亲身高的标准差。

双亲身高的标准差约为 1.8 英寸。 所以 0.5 英寸约为 0.28 个标准差。

```
sd_midparent = np.std(heights.column(0))
sd_midparent
1.8014050969207571
0.5/sd_midparent
0.27756111096536701
```

现在我们准备修改我们的预测函数，来预测标准单位。 所有改变的是，我们正在使用标准单位的值的表格，并定义如上所述的“接近”。

```
def predict_child_su(mpht_su):
    """Return a prediction of the height (in standard units) of a child
    whose parents have a midparent height of mpht_su in standard units.
    """
    close = 0.5/sd_midparent
    close_points = heights_SU.where('MidParent SU', are.between(mpht_su-close, mpht_su
 + close))
    return close_points.column('Child SU').mean()
heights_with_su_predictions = heights_SU.with_column(
    'Prediction SU', heights_SU.apply(predict_child_su, 'MidParent SU')
    )
heights_with_su_predictions.scatter('MidParent SU')
```

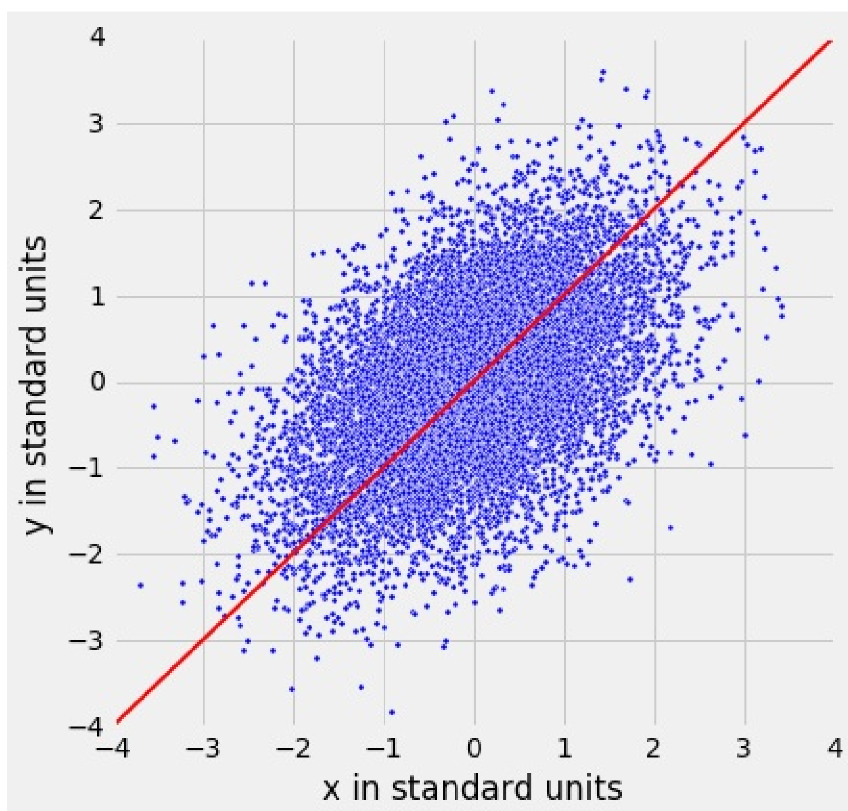


这个绘图看起来就像在原始刻度上绘图。 只改变了轴上的数字。 这证实了我们可以通过在标准单位下工作，来理解预测过程。

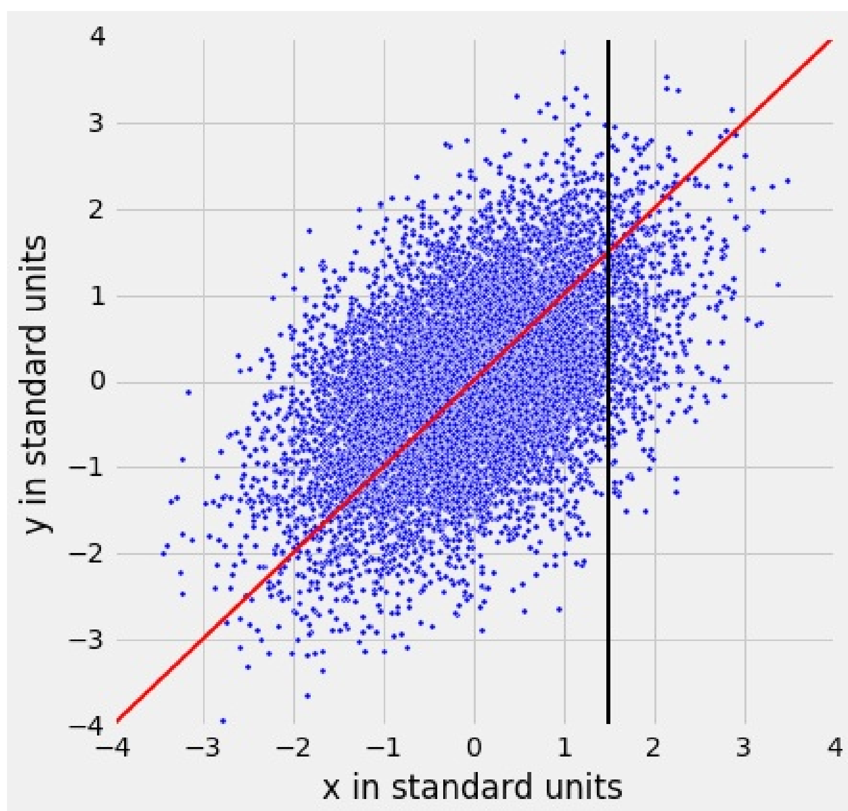
确定标准单位下的直线

高尔顿的散点图形状是个橄榄球 - 就是说，像橄榄球一样大致椭圆形。不是所有的散点图都是橄榄形的，甚至那些线性关联的也不都是。但在这一节中，我们假装我们是高尔顿，只能处理橄榄形的散点图。在下一节中，我们将把我们的分析推广到其他形状的绘图。

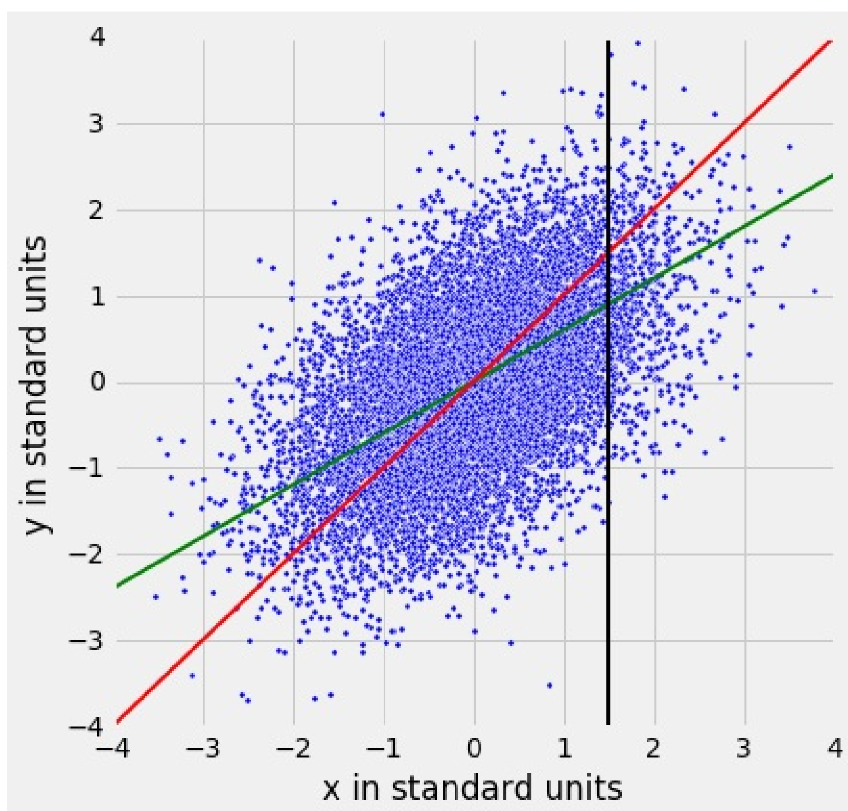
这里是一个橄榄形散点图，两个变量以标准单位测量。 45 度线显示为红色。



但是 45 度线不是经过垂直条形的中心的线。你可以看到在下图中，1.5 个标准单位的垂直线显示为黑色。蓝线附近的散点图上的点的高度都大致在 -2 到 3 的范围内。红线太高，无法命中中心。



所以 45 度线不是“均值图”。该线是下面显示的绿线。



两条线都经过原点`(0,0)`。绿线穿过垂直条形的中心（至少大概），比红色的 45 度线平坦。

45 度线的斜率为 1。所以绿色的“均值图”直线的斜率是正值但小于 1。

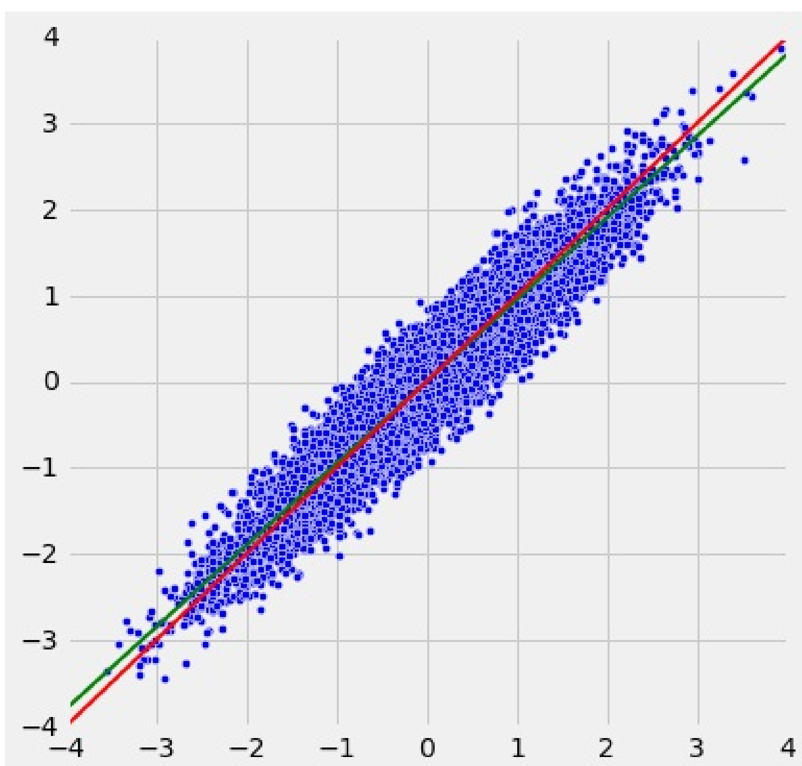
这可能是什么值呢？你猜对了 - 这是`r`。

标准单位下的回归直线

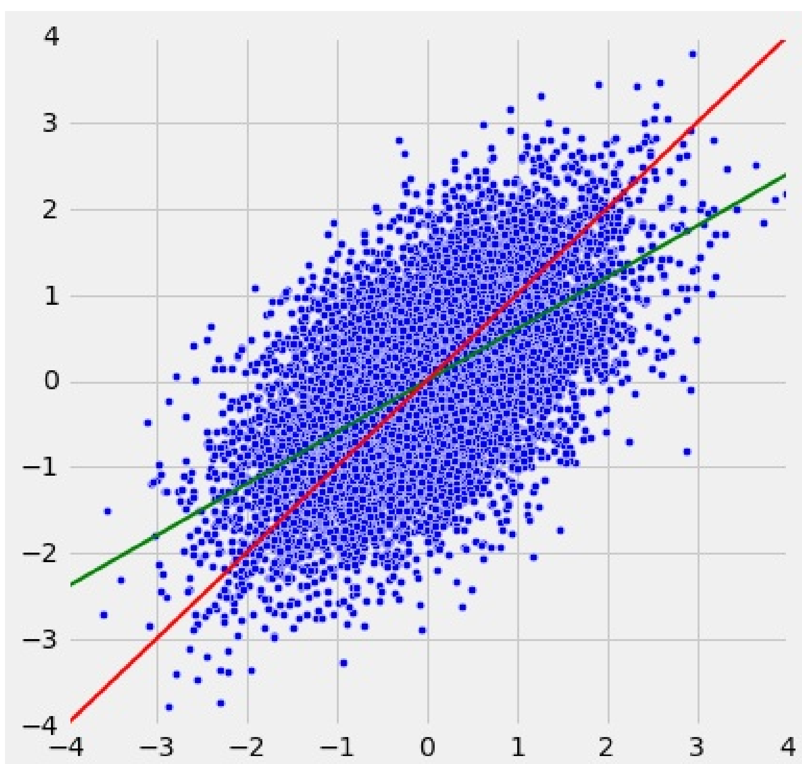
绿色的“均值图”线被称为回归直线，我们将很快解释原因。 但首先，让我们模拟一些`r`值不同的橄榄形散点图，看看直线是如何变化的。 在每种情况中，绘制红色 45 度线作比较。

执行模拟的函数为`regression_line`，并以`r`为参数。

```
regression_line(0.95)
```




```
regression_line(0.6)
```



当 `r` 接近于 1 时，散点图，45 度线和回归线都非常接近。但是对于 `r` 较低值来说，回归线显然更平坦。

回归效应

就预测而言，这意味着，对于双亲身高为 1.5 个标准单位的家长来说，我们对女子身高的预测要稍低于 1.5 个标准单位。如果双亲高度是 2 个标准单位，我们对子女身高的预测，会比 2 个标准单位少一些。

换句话说，我们预测，子女会比父母更接近均值。

弗朗西斯·高尔顿爵士就不高兴了。他一直希望，特别高的父母会有特别高的子女。然而，数据是清楚的，高尔顿意识到，高个子父母通常拥有并不是特别高的子女。高尔顿沮丧地将这种现象称为“回归平庸”。

高尔顿还注意到，特别矮的父母通常拥有相对于他们这一代高一些的子女。一般来说，一个变量的平均值远远低于另一个变量的平均值。这被称为回归效应。

回归直线的方程

在回归中，我们使用一个变量（我们称 `x` ）的值来预测另一个变量的值（我们称之为 `y` ）。当变量 `x` 和 `y` 以标准单位测量时，基于 `x` 预测 `y` 的回归线斜率为 `r` 并通过原点。 因此，回归线的方程可写为：

$$\text{estimate of } y ~=~ r \cdot x ~~~ \text{when both variables are measured in standard units}$$

在数据的原始单位下，就变成了：

$$\frac{\text{estimate of } y ~-~ \text{average of } y}{\text{SD of } y} ~=~ r \times \frac{\text{the given } x ~-~ \text{average of } x}{\text{SD of } x}$$

原始单位的回归线的斜率和截距可以从上图中导出。

$$\text{slope of the regression line} ~=~ r \cdot \frac{\text{SD of } y}{\text{SD of } x}$$

$$\text{intercept of the regression line} ~=~ \text{average of } y ~-~ \text{slope} \cdot \text{average of } x$$

下面的三个函数计算相关性，斜率和截距。 它们都有三个参数：表的名称，包含 `x` 的列的标签以及包含 `y` 的列的标签。

```
def correlation(t, label_x, label_y):
    return np.mean(standard_units(t.column(label_x))*standard_units(t.column(label_y))
)

def slope(t, label_x, label_y):
    r = correlation(t, label_x, label_y)
    return r*np.std(t.column(label_y))/np.std(t.column(label_x))

def intercept(t, label_x, label_y):
    return np.mean(t.column(label_y)) - slope(t, label_x, label_y)*np.mean(t.column(la
bel_x))
```

回归直线和高尔顿的数据

双亲身高和子女身高之间的相关性是 0.32：

```
galton_r = correlation(heights, 'MidParent', 'Child')
galton_r
0.32094989606395924
```

我们也可以找到回归直线的方程，来基于双亲身高预测子女身高：

```
galton_slope = slope(heights, 'MidParent', 'Child')
galton_intercept = intercept(heights, 'MidParent', 'Child')
galton_slope, galton_intercept
(0.63736089696947895, 22.636240549589751)
```

回归直线的方程是：

$$\text{estimate of child's height} ~=~ 0.64 \cdot \text{midparent height} ~+~ 22.64$$

这也成为回归方程。回归方程的主要用途是根据 `x` 预测 `y`。

例如，对于 70.48 英寸的双亲身高，回归直线预测，子女身高为 67.56 英寸。

```
galton_slope*70.48 + galton_intercept
67.557436567998622
```

我们最初的预测，通过计算双亲身高接近 70.48 的所有子女的平均身高来完成，这个预测非常接近：67.63 英寸，而回归线的预测是 67.55 英寸。

```
heights_with_predictions.where('MidParent', are.equal_to(70.48)).show(3)
```

MidParent	Child	Prediction
70.48	74	67.6342
70.48	70	67.6342
70.48	68	67.6342

（省略了 5 行）

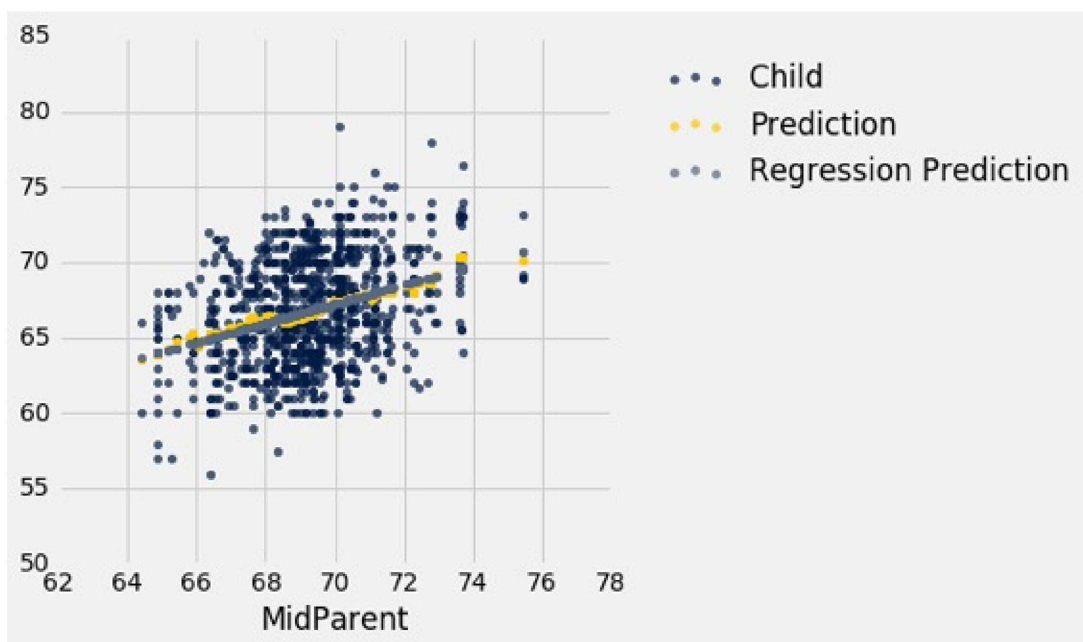
这里是高尔顿的表格的所有行，我们的原始预测，以及子女身高的回归预测。

```
heights_with_predictions = heights_with_predictions.with_column(
    'Regression Prediction', galton_slope*heights.column('MidParent') + galton_interce
pt
)
heights_with_predictions
```

MidParent	Child	Prediction	Regression Prediction
75.43	73.2	70.1	70.7124
75.43	69.2	70.1	70.7124
75.43	69	70.1	70.7124
75.43	69	70.1	70.7124
73.66	73.5	70.4158	69.5842
73.66	72.5	70.4158	69.5842
73.66	65.5	70.4158	69.5842
73.66	65.5	70.4158	69.5842
72.06	71	68.5025	68.5645
72.06	68	68.5025	68.5645

（省略了 924 行）

```
heights_with_predictions.scatter('MidParent')
```



灰色圆点显示回归预测，全部在回归线上。 注意这条线与均值的金色图非常接近。 对于这些数据，回归线很好地逼近垂直条形的中心。

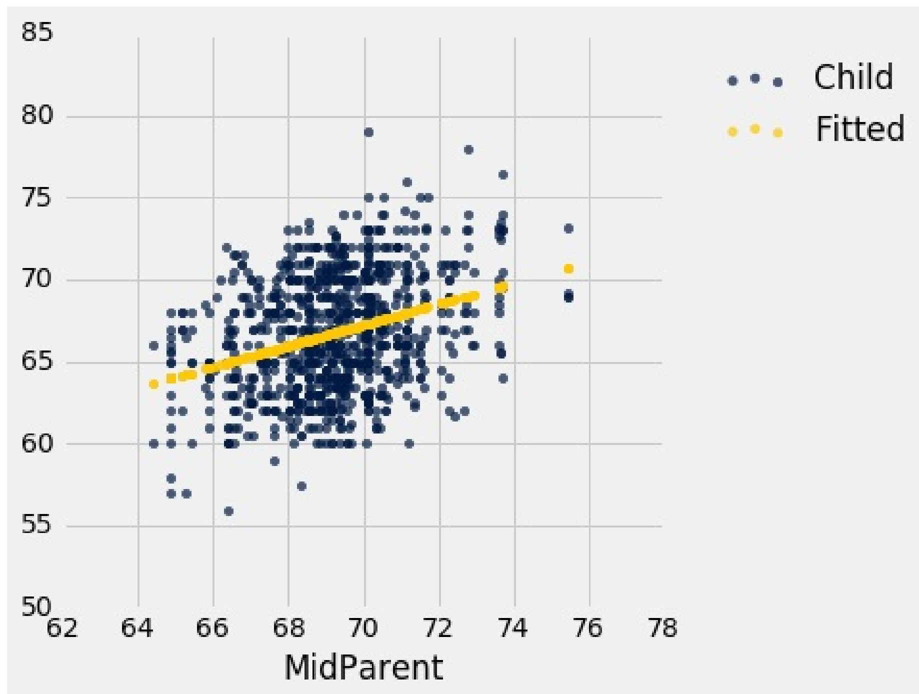
拟合值

所有的预测值都在直线上，被称为“拟合值”。 函数 `fit` 使用表名和 `x` 和 `y` 的标签，并返回一个拟合值数组，散点图中每个点一个。

```
def fit(table, x, y):
    """Return the height of the regression line at each x value."""
    a = slope(table, x, y)
    b = intercept(table, x, y)
    return a * table.column(x) + b
```

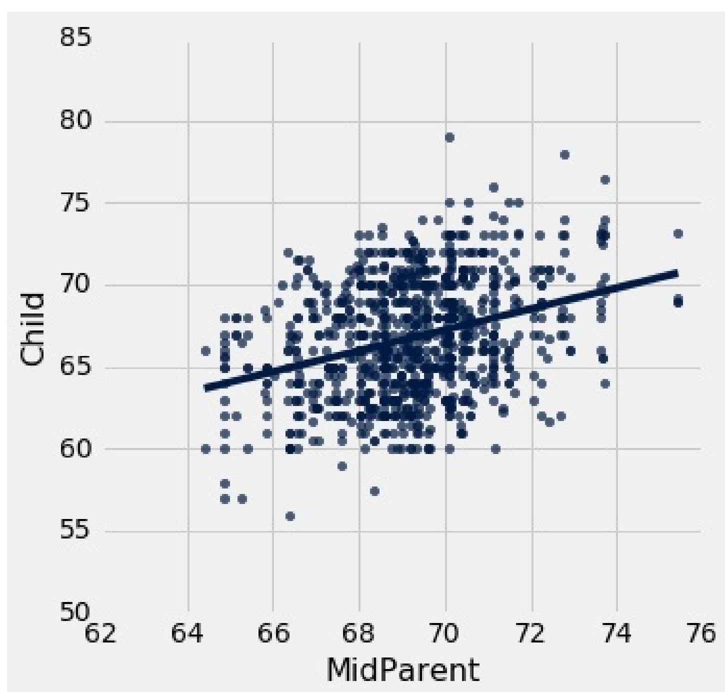
下图比上图更轻易看到直线：

```
heights.with_column('Fitted', fit(heights, 'MidParent', 'Child')).scatter('MidParent')
```



另一个绘制直线的方式是在表方法 `scatter` 中，使用选项 `fit_line=True`。

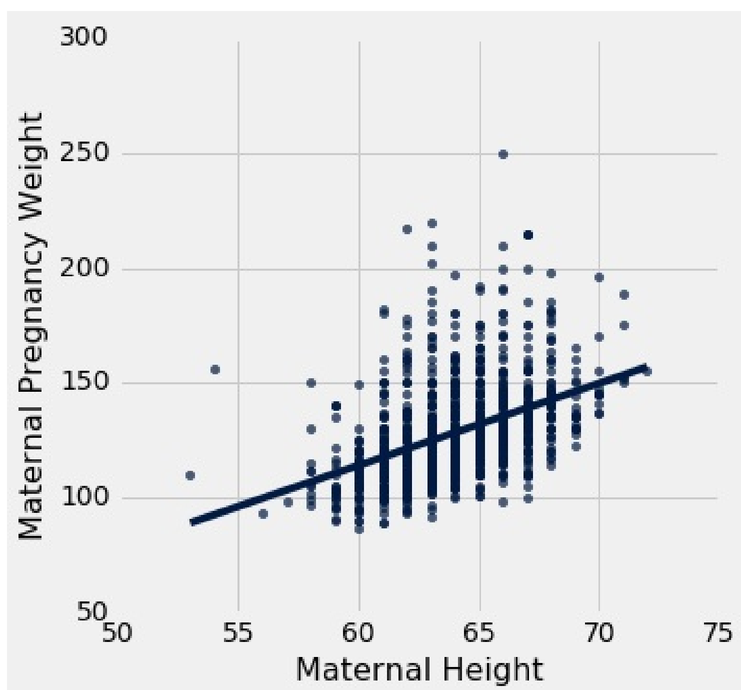
```
heights.scatter('MidParent', fit_line=True)
```



斜率的测量单位

斜率是一个比值，值得花点时间来研究它的测量单位。 我们的例子来自熟悉的医院系统中产妇的数据集。 孕期体重与高度的散点图看起来像是一个橄榄球，已经在一场比赛中使用了很多次，但足够接近橄榄球，我们可以让我们的拟合直线穿过它来证明。 在后面的章节中，我们将看到如何使这种证明更正式。

```
baby = Table.read_table('baby.csv')
baby.scatter('Maternal Height', 'Maternal Pregnancy Weight', fit_line=True)
```



```
slope(baby, 'Maternal Height', 'Maternal Pregnancy Weight')
3.5728462592750558
```

回归线的斜率是 3.57 磅每英寸。 这意味着，对于身高相差 1 英寸的两名女性来说，我们对孕期体重的预测相差 3.57 磅。 对于身高相差 2 英寸的女性，我们预测的孕期体重相差 `2 * 3.57 ~= 7.14` 磅。

请注意，散点图中的连续垂直条形相距 1 英寸，因为高度已经舍入到最近的英寸。 另一种考虑斜率的方法是取两个相连的条形（相隔 1 英寸），相当于两组身高相差 1 英寸的女性。 3.57 磅每英寸的斜率意味着，较高组的平均孕期体重比较矮组多大约 3.57 磅。

示例

假设我们的目标是使用回归，基于巴塞特猎犬的体重来估计它的身高，所用的样本与回归模型看起来一致。 假设观察到的相关性 `r` 为 0.5，并且这两个变量的汇总统计量如下表所示：

	average	SD
height	14 inches	2 inches
weight	50 pounds	5 pounds

为了计算回归线的方程，我们需要斜率和截距。

$$\mathrm{slope} ~=~ \frac{r \cdot \mathrm{SD\ of}\ y}{\mathrm{SD\ of}\ x} ~=~ \frac{0.5 \cdot 2\ \mathrm{inches}}{5\ \mathrm{pounds}} ~=~ 0.2\ \mathrm{inches\ per\ pound}$$

$$\mathrm{intercept} ~=~ \mathrm{average\ of}\ y - \mathrm{slope}\cdot \mathrm{average\ of}\ x ~=~ 14\ \mathrm{inches} ~-~ 0.2\ \mathrm{inches\ per\ pound} \cdot 50\ \mathrm{pounds} ~=~ 4\ \mathrm{inches}$$

回归线的方程允许我们，根据给定重量（磅）计算估计高度（英寸）：

$$\mathrm{estimated\ height} ~=~ 0.2 \cdot \mathrm{given\ weight} ~+~ 4$$

线的斜率衡量随着重量的单位增长的估计高度的增长。 斜率是正值，重要的是要注意，这并不表示我们认为，如果体重增加巴塞特猎狗就会变得更高。 斜率反映了两组狗的平均身高的差异，这两组狗的体重相差 1 磅。 具体来说，考虑一组重量为 `w` 磅，以及另一组重量为 `w + 1` 磅的狗。 我们估计，第二组的均值高出 0.2 英寸。 对于样本中的所有 `w` 值都是如此。

一般来说，回归线的斜率可以解释为随着 `x` 单位增长的 `y` 平均增长。 请注意，如果斜率为负值，那么对于 `x` 的每单位增长，`y` 的平均值会减少。

尾注

即使我们没有建立回归方程的数学基础，我们可以看到，当散点图是橄榄形的时候，它会给出相当好的预测。 这是一个令人惊讶的数学事实，无论散点图的形状如何，同一个方程给出所有直线中的“最好”的预测。 这是下一节的主题。

最小二乘法

我们已经回溯了高尔顿和皮尔森用于开发回归线方程的步骤，它穿过橄榄形的散点图。但不是所有的散点图都是橄榄形的，甚至不是线性的。每个散点图都有一个“最优”直线吗？如果是这样，我们仍然可以使用上一节中开发的斜率和截距公式，还是需要新的公式？

为了解决这些问题，我们需要一个“最优”的合理定义。回想一下，这条线的目的是预测或估计 `y` 的值，在给定 `x` 值的情况下。估计通常不是完美的。每个值都由于误差而偏离真正的值。“最优”直线的合理标准是，它在所有直线中总体误差尽可能最小。

在本节中，我们将精确确定这个标准，看看我们能否确定标准下的最优直线。

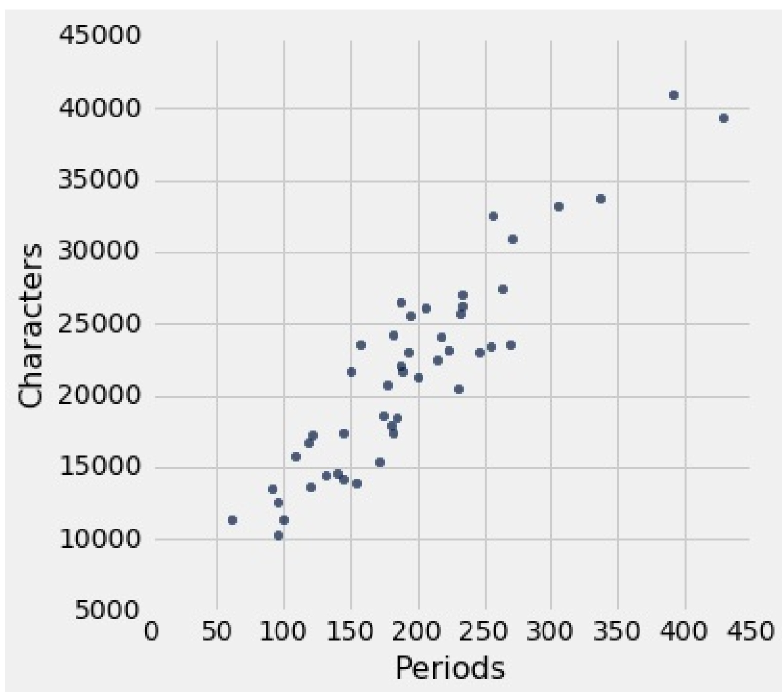
我们的第一个例子是小说《小女人》数据集，每章都有一行。目标是根据句子数来估计字符数（即字母，空格标点符号等等）。回想一下，我们在本课程的第一堂课中试图实现它。

```
little_women = Table.read_table('little_women.csv')
little_women = little_women.move_to_start('Periods')
little_women.show(3)
```

Periods	Characters
189	21759
188	22148
231	20558

（省略了 44 行）

```
little_women.scatter('Periods', 'Characters')
```



为了探索数据，我们将需要使用上一节定义的函数 `correlation` ， `slope` ， `intercept` 和 `fit` 。

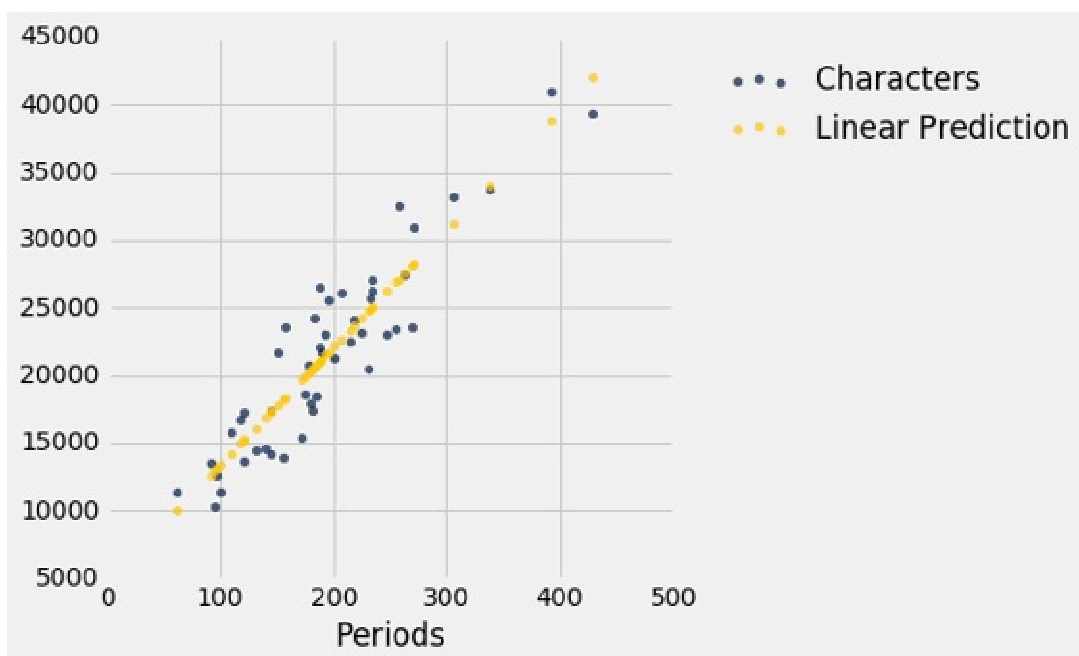
```
correlation(little_women, 'Periods', 'Characters')
0.92295768958548163
```

散点图明显接近线性，相关性大于 0.92。

估计中的误差

下图显示了我们在上一节中开发的散点图和直线。 我们还不知道这是否是所有直线中最优的。 我们首先必须准确表达“最优”的意思。

```
lw_with_predictions = little_women.with_column('Linear Prediction', fit(little_women,
'Periods', 'Characters'))
lw_with_predictions.scatter('Periods')
```



对应于散点图上的每个点，预测的误差是计算为实际值减去预测值。 它是点与直线之间的垂直距离，如果点在线之下，则为负值。

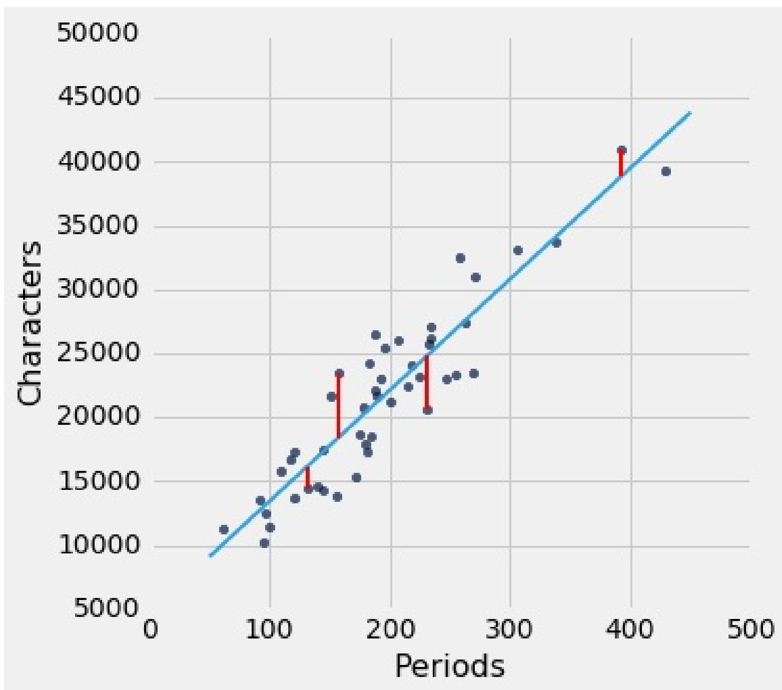
```
actual = lw_with_predictions.column('Characters')
predicted = lw_with_predictions.column('Linear Prediction')
errors = actual - predicted
lw_with_predictions.with_column('Error', errors)
```

Periods	Characters	Linear Prediction	Error
189	21759	21183.6	575.403
188	22148	21096.6	1051.38
231	20558	24836.7	-4278.67
195	25526	21705.5	3820.54
255	23395	26924.1	-3529.13
140	14622	16921.7	-2299.68
131	14431	16138.9	-1707.88
214	22476	23358	-882.043
337	33767	34056.3	-289.317
185	18508	20835.7	-2327.69

（省略了 37 行）

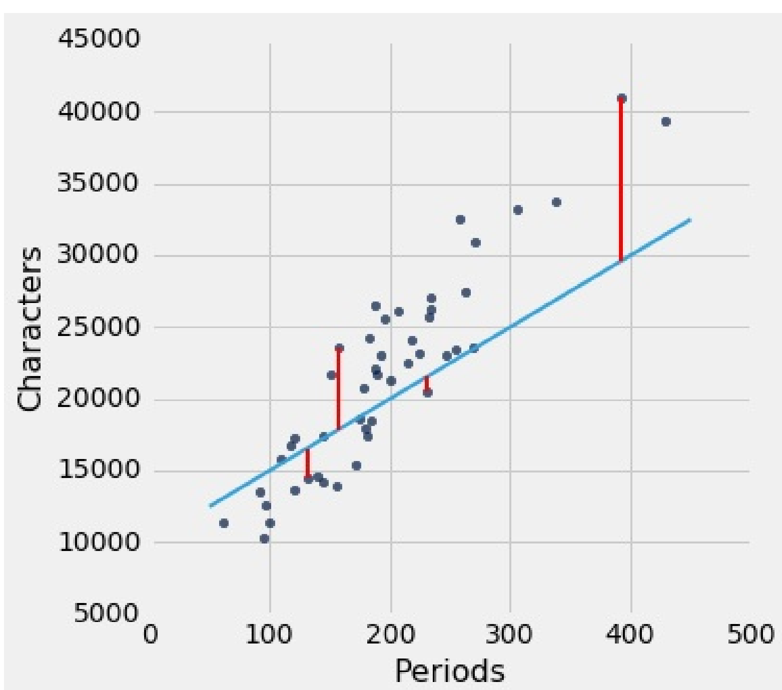
我们可以使用 `slope` 和 `intercept` 来计算拟合直线的斜率和截距。 下图显示了该直线（浅蓝色）。 对应于四个点的误差以红色显示。 这四个点没什么特别的。 他们只是为了展示的清晰而被选中。 函数 `lw_errors` 以斜率和截距（按照该顺序）作为参数，并绘制该图形。

```
lw_reg_slope = slope(little_women, 'Periods', 'Characters')
lw_reg_intercept = intercept(little_women, 'Periods', 'Characters')
print('Slope of Regression Line:    ', np.round(lw_reg_slope), 'characters per period'
)
print('Intercept of Regression Line:', np.round(lw_reg_intercept), 'characters')
lw_errors(lw_reg_slope, lw_reg_intercept)
Slope of Regression Line:     87.0 characters per period
Intercept of Regression Line: 4745.0 characters
```

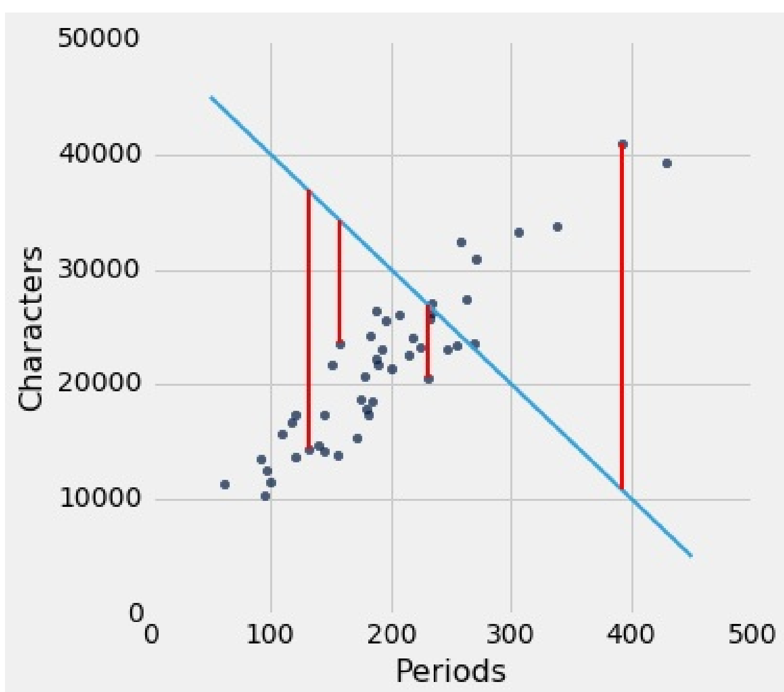


如果我们用不同的线来创建我们的估计，误差将会不同。 下面的图表显示了如果我们使用另一条线进行估算，误差会有多大。 第二张图显示了通过使用完全愚蠢的线获得了较大误差。

```
lw_errors(50, 10000)
```



```
lw_errors(-100, 50000)
```



均方根误差（RMSE）

我们现在需要的是误差大小的一个总体衡量。 你会认识到创建它的方法 - 这正是我们开发标准差的方式。

如果你使用任意直线来计算你的估计值，那么你的一些误差可能是正的，而其他的则是负的。 为了避免误差大小在测量时抵消，我们将采用误差平方的均值而不是误差的均值。

估计的均方误差大概是误差的平方有多大，但正如我们前面提到的，它的单位很难解释。 取平方根产生均方根误差（RMSE），与预测变量的单位相同，因此更容易理解。

使 RMSE 最小

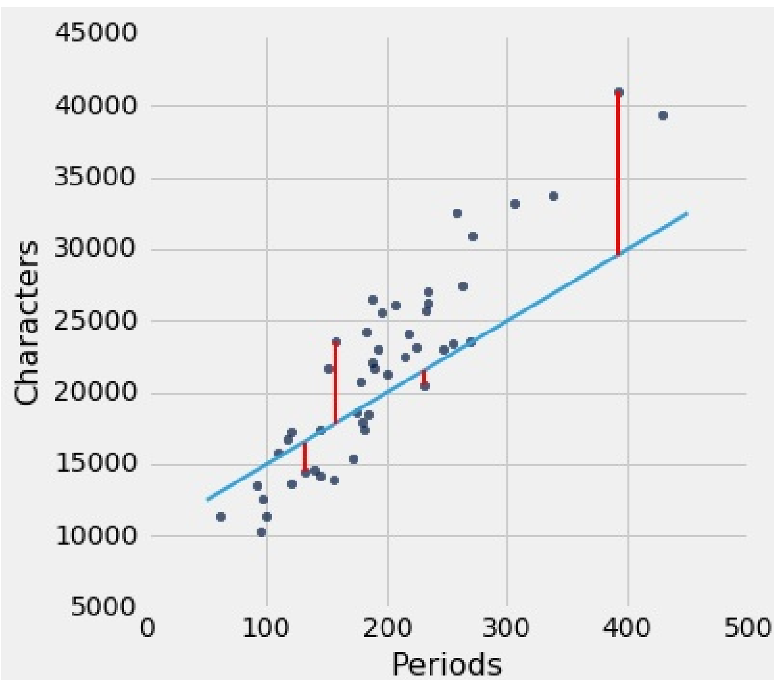
到目前为止，我们的观察可以总结如下。

- 要根据 `x` 估算 `y`，可以使用任何你想要的直线。
- 每个直线都有估计的均方根误差。
- “更好”的直线有更小的误差。

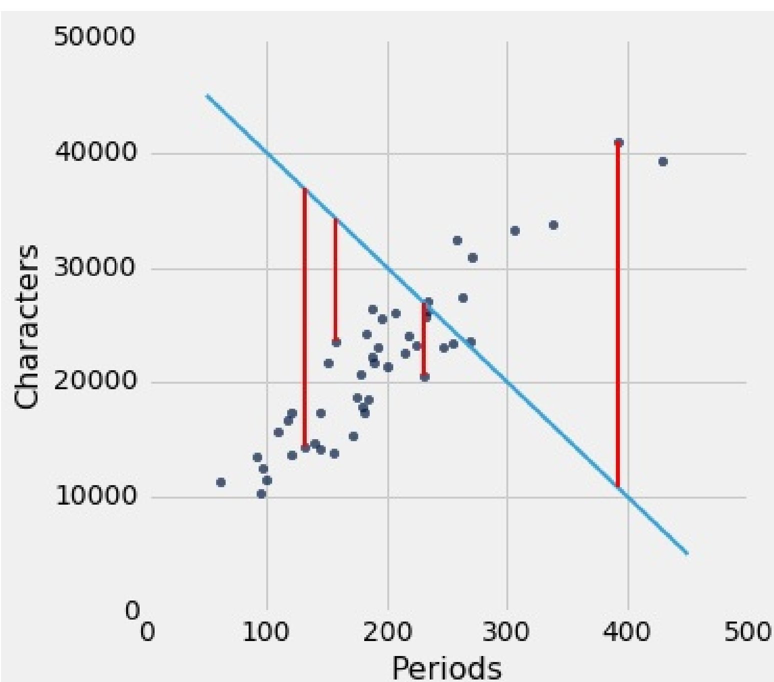
有没有“最好”的直线？ 也就是说，是否有一条线可以使所有行中的均方根误差最小？

为了回答这个问题，我们首先定义一个函数 `lw_rmse`，通过《小女人》的散点图来计算任意直线的均方根误差。 函数将斜率和截距（按此顺序）作为参数。

```
def lw_rmse(slope, intercept):
    lw_errors(slope, intercept)
    x = little_women.column('Periods')
    y = little_women.column('Characters')
    fitted = slope * x + intercept
    mse = np.mean((y - fitted) ** 2)
    print("Root mean squared error:", mse ** 0.5)
lw_rmse(50, 10000)
Root mean squared error: 4322.16783177
```

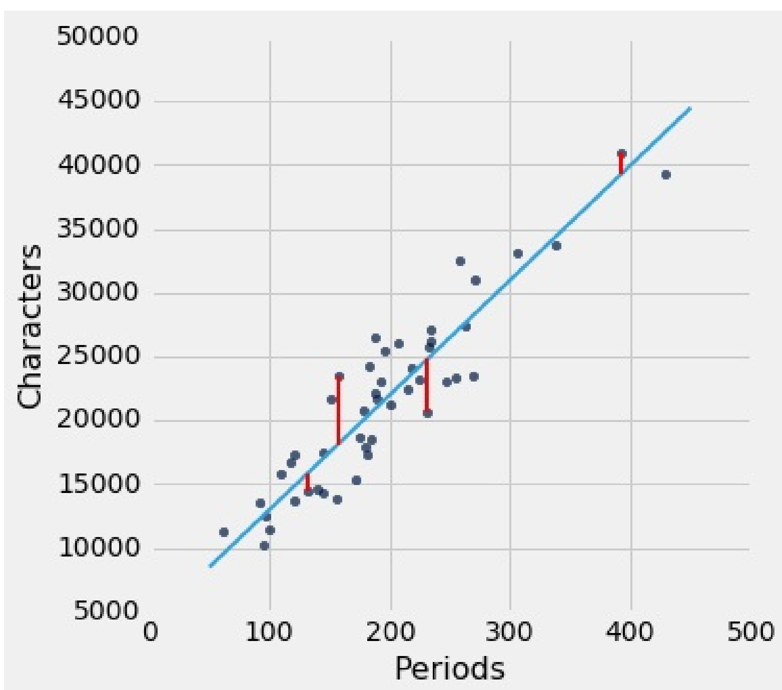



```
lw_rmse(-100, 50000)
Root mean squared error: 16710.1198374
```



正如预期的那样，不好的直线 RMSE 很大。 但是如果我们选择接近于回归线的斜率和截距，则 RMSE 要小得多。

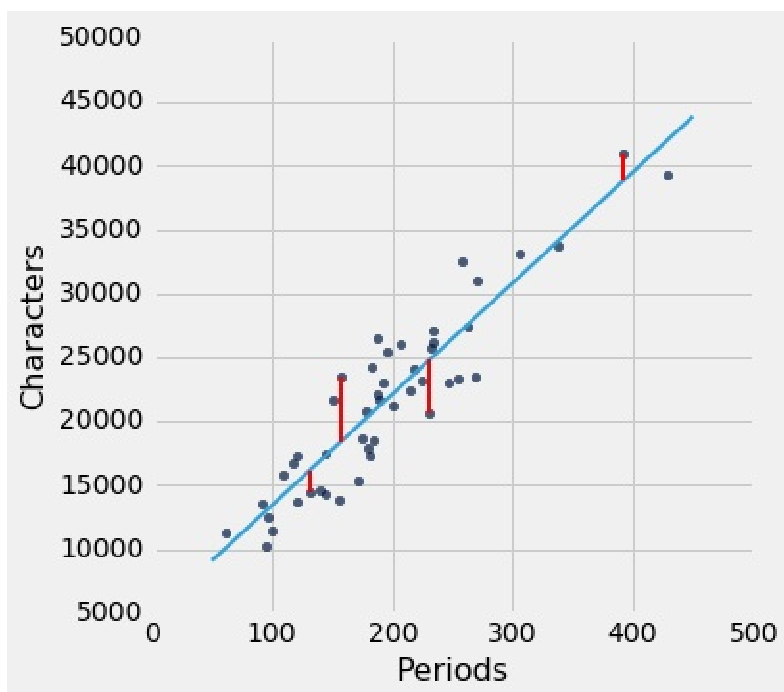
```
lw_rmse(90, 4000)
Root mean squared error: 2715.53910638
```



这是对应于回归线的均方根误差。 通过显着的数学事实，没有其他线路能击败这一条。

回归线是所有直线之间的唯一直线，使估计的均方误差最小。

```
lw_rmse(lw_reg_slope, lw_reg_intercept)
Root mean squared error: 2701.69078531
```



这个声明的证明需要超出本课程范围的抽象数学。 另一方面，我们有一个强大的工具 -- Python，它可以轻松执行大量的数值计算。 所以我们可以使用 Python 来确认回归线最小化的均方误差。

数值优化

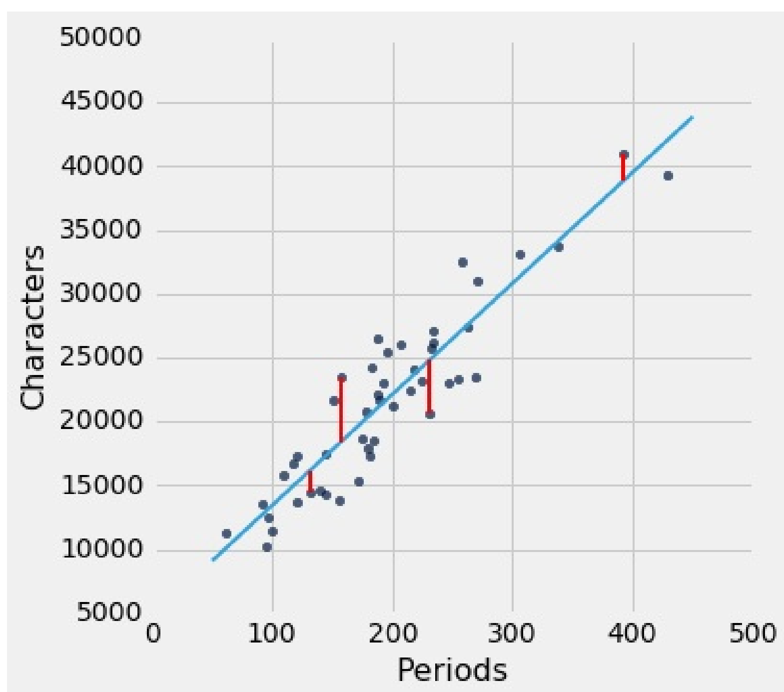
首先注意，使均方根误差最小的直线，也是使平方误差最小的直线。 平方根对最小值没有任何影响。 所以我们会为自己节省一个计算步骤，并将平均方差 MSE 减到最小。

我们试图根据《小女人》的句子数（`x`）来预测字符数量（`y`）。 如果我们使用 $\text{prediction} = ax + b$ 直线，它将有一个 MSE，它取决于斜率 `a` 和截距 `b`。 函数 `lw_mse` 以斜率和截距为参数，并返回相应的 MSE。

```
def lw_mse(any_slope, any_intercept):
    x = little_women.column('Periods')
    y = little_women.column('Characters')
    fitted = any_slope*x + any_intercept
    return np.mean((y - fitted) ** 2)
```

让我们确认一下， `lw_mse` 得到回归线的 RMSE 的正确答案。 请记住， `lw_mse` 返回均方误差，所以我们必须取平方根来得到 RMSE。

```
lw_mse(lw_reg_slope, lw_reg_intercept)**0.5
2701.690785311856
```



它和我们之前使用 `lw_rmse` 得到的值相同。

```
lw_rmse(lw_reg_slope, lw_reg_intercept)
Root mean squared error: 2701.69078531
```

你可以确认对于其他的斜率和截距，`lw_mse` 也返回正确的值。 例如，这里是我们之前尝试的，非常不好的直线的 RMSE。

```
lw_mse(-100, 50000)**0.5
16710.119837353752
```

这里是这条直线的 RMSE，它接近回归线。

```
lw_mse(90, 4000)**0.5
2715.5391063834586
```

如果我们尝试不同的值，我们可以通过反复试验找到一个误差较低的斜率和截距，但这需要一段时间。 幸运的是，有一个 Python 函数为我们做了所有的试错。

`minimize` 函数可用于寻找函数的参数，函数在这里返回其最小值。 Python 使用类似的试错法，遵循使输出值递减的变化量。

`minimize` 的参数是一个函数，它本身接受数值参数并返回一个数值。 例如，函数 `lw_mse` 以数值斜率和截距作为参数，并返回相应的 MSE。

调用 `minimize(lw_mse)` 返回一个数组，由斜率和截距组成，它们使 MSE 最小。 这些最小值是通过智能试错得出的极好的近似值，而不是基于公式的精确值。

```
best = minimize(lw_mse)
best
array([   86.97784117,   4744.78484535])
```

这些值与我们之前使用`slope`和`intercept`函数计算的值相同。 由于最小化的不精确性，我们看到较小的偏差，但是这些值本质上是相同的。

```
print("slope from formula:        ", lw_reg_slope)
print("slope from minimize:       ", best.item(0))
print("intercept from formula:    ", lw_reg_intercept)
print("intercept from minimize:   ", best.item(1))
slope from formula:         86.9778412583
slope from minimize:        86.97784116615884
intercept from formula:     4744.78479657
intercept from minimize:    4744.784845352655
```

最小二乘直线

因此我们发现，不仅回归线具有最小的均方误差，而且均方误差的最小化也给出了回归线。 回归线是最小化均方误差的唯一直线。

这就是回归线有时被称为“最小二乘直线”的原因。

最小二乘回归

在前面的章节中，我们开发了回归直线的斜率和截距方程，它穿过一个橄榄形的散点图。 事实证明，无论散点图的形状如何，最小二乘直线的斜率和截距都与我们开发的公式相同。

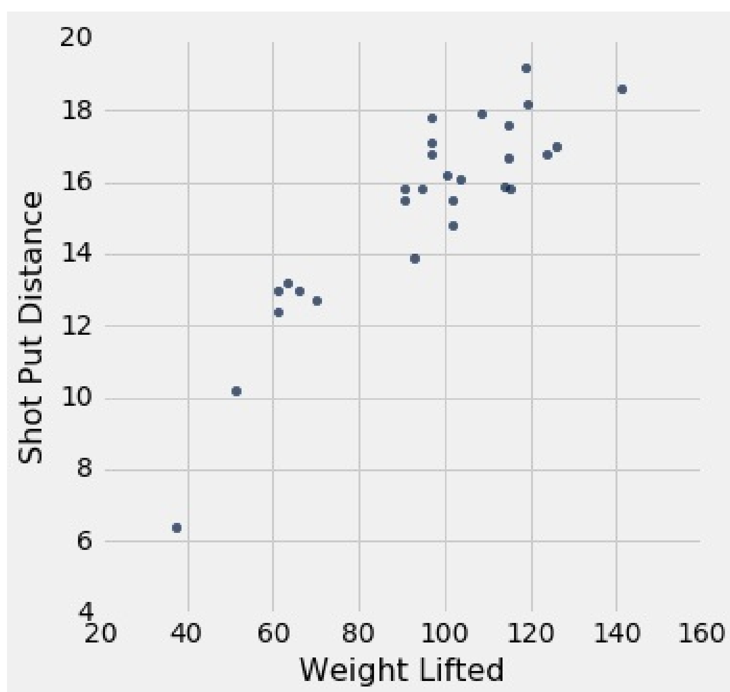
我们在《小女人》的例子中看到了它，但是让我们以散点图显然不是橄榄形的例子来证实它。 对于这些数据，我们再次受惠于佛罗里达大学 Larry Winner 教授的丰富数据档案。 《国际运动科学杂志》（International Journal of Exercise Science）2013 年的一项研究，研究了大学生铅球运动员，并考察了力量与铅球距离的关系。 总体由 28 名女大学生运动员组成。 运动员在赛季前的“1RM power clean”中举起的最大值（公斤）是衡量力量的指标。 距离（米）是运动员个人最佳成绩。

```
shotput = Table.read_table('shotput.csv')
shotput
```

Weight Lifted	Shot Put Distance
37.5	6.4
51.5	10.2
61.3	12.4
61.3	13
63.6	13.2
66.1	13
70	12.7
92.7	13.9
90.5	15.5
90.5	15.8

（省略了 18 行）

```
shotput.scatter('Weight Lifted')
```



这不是橄榄形的散点图。事实上，它似乎有一点非线性成分。但是，如果我们坚持用一条直线来做出预测，那么所有直线之中仍然有一条最好的直线。

我们为回归线的斜率和截距建立公式，它来源于橄榄形的散点图，并给出了下列值：

```
slope(shotput, 'Weight Lifted', 'Shot Put Distance')
0.098343821597819972
intercept(shotput, 'Weight Lifted', 'Shot Put Distance')
5.9596290983739522
```

即使散点图不是橄榄形，使用这些公式还有意义吗？我们可以通过求出使 MSE 最小的斜率和截距来回答这个问题。

我们将定义函数 `shotput_linear_mse`，以斜体和截距作为参数并返回相应的 MSE。然后将 `minimize` 应用于 `shotput_linear_mse` 将返回最优斜率和截距。

```
def shotput_linear_mse(any_slope, any_intercept):
    x = shotput.column('Weight Lifted')
    y = shotput.column('Shot Put Distance')
    fitted = any_slope*x + any_intercept
    return np.mean((y - fitted) ** 2)
minimize(shotput_linear_mse)
array([ 0.09834382,  5.95962911])
```

这些值与我们使用我们的公式得到的值相同。总结：

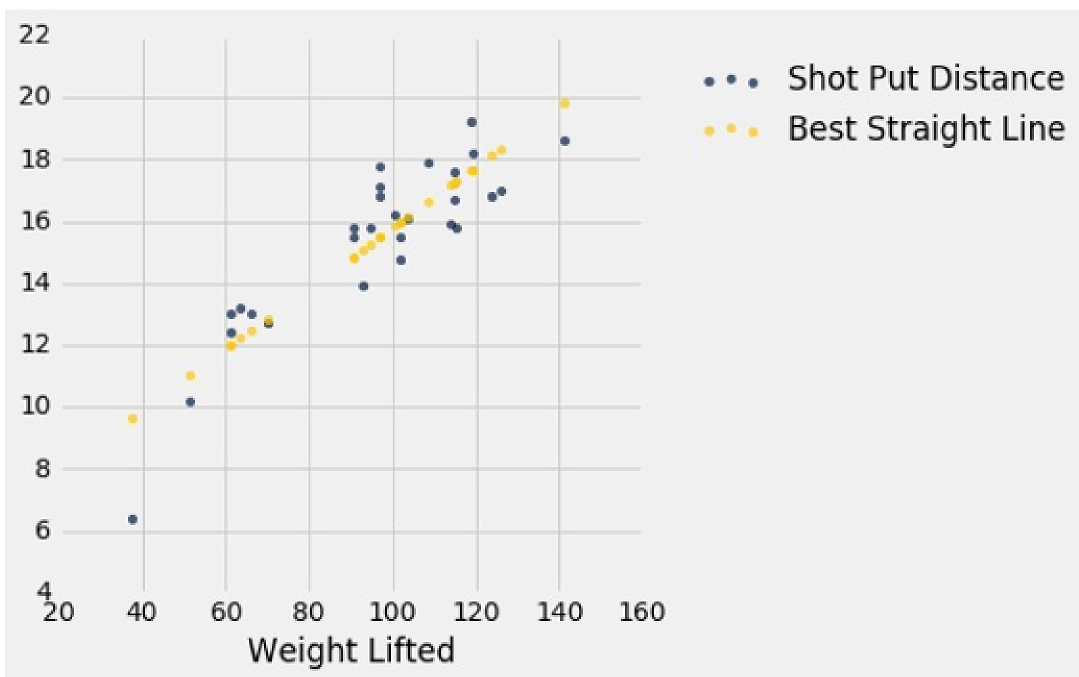
无论散点图的形状如何，都有一条独特的线，可以使估计的均方误差最小。它被称为回归线，其斜率和截距由下式给出：

$$\mathrm{slope\ of\ the\ regression\ line} ~=~ r \cdot \frac{\mathrm{SD\ of\ }y}{\mathrm{SD\ of\ }x}$$

> 译者注：也就是 `cov(x, y)/var(x)`。

$$\mathrm{intercept\ of\ the\ regression\ line} ~=~ \mathrm{average\ of\ }y ~-~ \mathrm{slope} \cdot \mathrm{average\ of\ }x$$

```
fitted = fit(shotput, 'Weight Lifted', 'Shot Put Distance')
shotput.with_column('Best Straight Line', fitted).scatter('Weight Lifted')
```



非线性回归

上面的图表强化了我们之前的观察，即散点图有点弯曲。 因此，最好拟合曲线而不是直线。 研究假设举起的重量与铅球距离之间是二次关系。 所以让我们使用二次函数来预测，看看我们能否找到最好的曲线。

我们必须找到所有二次函数中最好的二次函数，而不是所有直线中最好的直线。 最小二乘法允许我们这样做。

这种最小化的数学是复杂的，不容易仅仅通过检查散点图来发现。 但是数值最小化和线性预测一样简单！ 再次通过使用最小化我们可以得到最好的二次预测。 让我们看看这是如何工作的。

回想一下，二次函数的形式：

```
f(x) = ax^2 + bx + c
```

`a` 、 `b` 和 `c` 是常数。

为了基于举起的重量找到最好的二次函数来预测距离，使用最小二乘法，我们首先编写一个函数，以三个常量为自变量的，用上面的二次函数计算拟合值，然后返回均方误差。

该函数被称为 `shotput_quadratic_mse` 。 请注意，定义与 `lw_mse` 的定义类似，不同的是拟合值基于二次函数而不是线性。

```
def shotput_quadratic_mse(a, b, c):
    x = shotput.column('Weight Lifted')
    y = shotput.column('Shot Put Distance')
    fitted = a*(x**2) + b*x + c
    return np.mean((y - fitted) ** 2)
```

我们现在可以像之前那样使用`minimize`，并找到使 MSE 最小的常数。

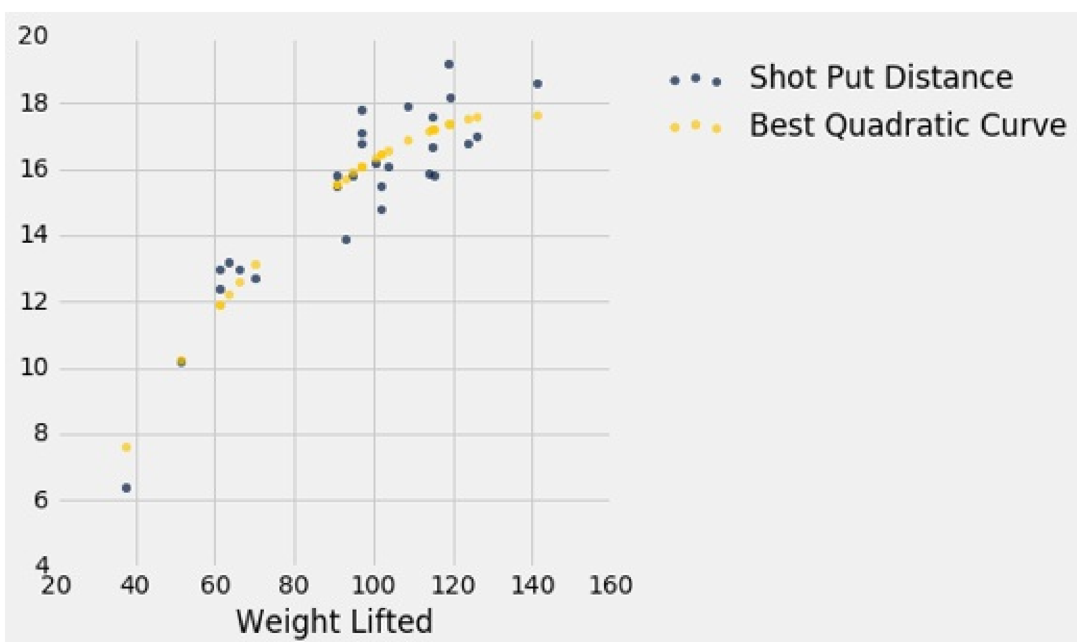
```
best = minimize(shotput_quadratic_mse)
best
array([ -1.04004838e-03,   2.82708045e-01,  -1.53182115e+00])
```

我们预测，一个举起`x`公斤的运动员的铅球距离大概是`-0.00104x^2 + 0.2827x - 1.5318`米。例如，如果运动员可以举起 100 公斤，预测的距离是 16.33 米。在散点图上，在 100 公斤左右的垂直条形的中心附近。

```
(-0.00104)*(100**2) + 0.2827*100 - 1.5318
16.3382
```

以下是所有`Weight Lifted`的预测。你可以看到他们穿过散点图的中心，大致上接近。

```
x = shotput.column(0)
shotput_fit = best.item(0)*(x**2) + best.item(1)*x + best.item(2)
shotput.with_column('Best Quadratic Curve', shotput_fit).scatter(0)
```



视觉诊断

假设数据科学家已经决定使用线性回归，基于预测变量估计响应变量的值。为了了解这种估计方法的效果如何，数据科学家必须知道估计值距离实际值多远。这些差异被称为残差。

$$\mathrm{residual} ~=~ \mathrm{observed ~ value} ~-~ \mathrm{regression ~ estimate}$$

残差就是剩下的东西 - 估计之后的剩余。

残差是回归线和点的垂直距离。散点图中的每个点都有残差。残差是 `y` 的观测值与 `y` 的拟合值之间的差值，所以对于点 `(x, y)` ：

$$\mathrm{residual} ~~ = ~~ y ~-~ \mathrm{fitted ~ value ~ of ~} y ~~ = ~~ y ~-~ \mathrm{height ~ of ~ regression ~ line ~ at ~} x$$

`residual` 函数计算残差。该计算假设我们已经定义的所有相关函数：`standard_units`，`correlation`，`slope`，`intercept` 和 `fit`。

```
def residual(table, x, y):
    return table.column(y) - fit(table, x, y)
```

继续使用高尔顿的数据的例子，基于双亲身高（预测变量）来估计成年子女身高（响应变量），让我们计算出拟合值和残差。

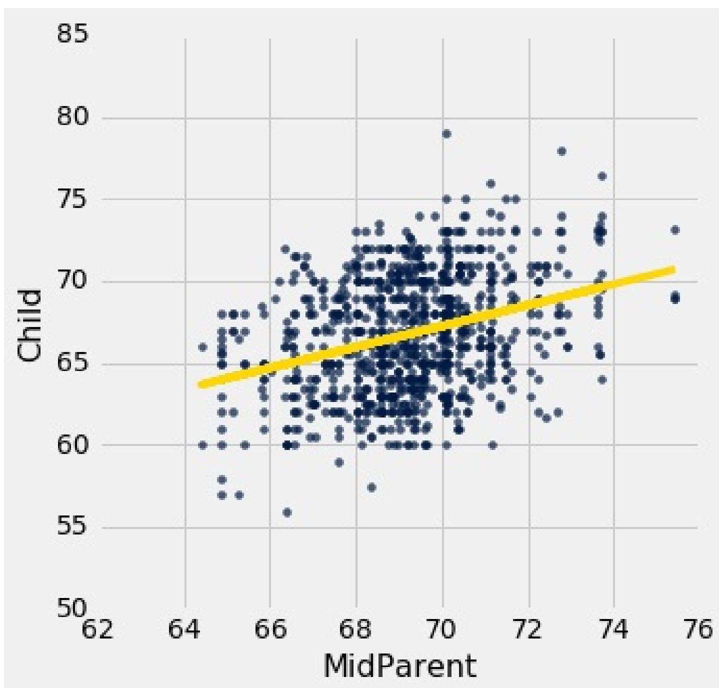
```
heights = heights.with_columns(
        'Fitted Value', fit(heights, 'MidParent', 'Child'),
        'Residual', residual(heights, 'MidParent', 'Child')
    )
heights
```

MidParent	Child	Fitted Value	Residual
75.43	73.2	70.7124	2.48763
75.43	69.2	70.7124	-1.51237
75.43	69	70.7124	-1.71237
75.43	69	70.7124	-1.71237
73.66	73.5	69.5842	3.91576
73.66	72.5	69.5842	2.91576
73.66	65.5	69.5842	-4.08424
73.66	65.5	69.5842	-4.08424
72.06	71	68.5645	2.43553
72.06	68	68.5645	-0.564467

（省略了 924 行）

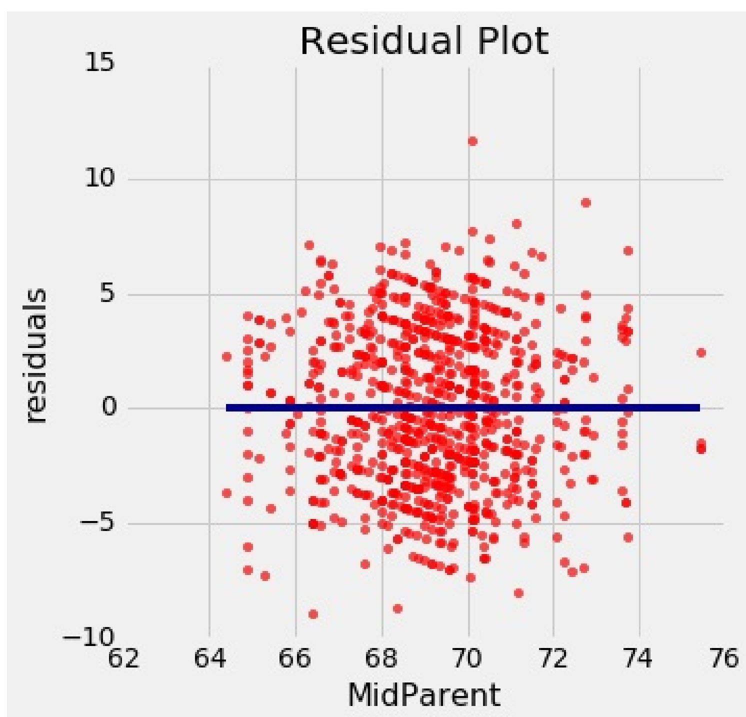
如果要处理的变量太多，以可视化开始总是很有帮助的。函数 `scatter_fit` 绘制数据的散点图，以及回归线。

```
def scatter_fit(table, x, y):
    table.scatter(x, y, s=15)
    plots.plot(table.column(x), fit(table, x, y), lw=4, color='gold')
    plots.xlabel(x)
    plots.ylabel(y)
scatter_fit(heights, 'MidParent', 'Child')
```



通过绘制残差和预测变量来绘制残差图。函数 `residual_plot` 就是这样做的。

```
def residual_plot(table, x, y):
    x_array = table.column(x)
    t = Table().with_columns(
            x, x_array,
            'residuals', residual(table, x, y)
        )
    t.scatter(x, 'residuals', color='r')
    xlims = make_array(min(x_array), max(x_array))
    plots.plot(xlims, make_array(0, 0), color='darkblue', lw=4)
    plots.title('Residual Plot')
residual_plot(heights, 'MidParent', 'Child')
```

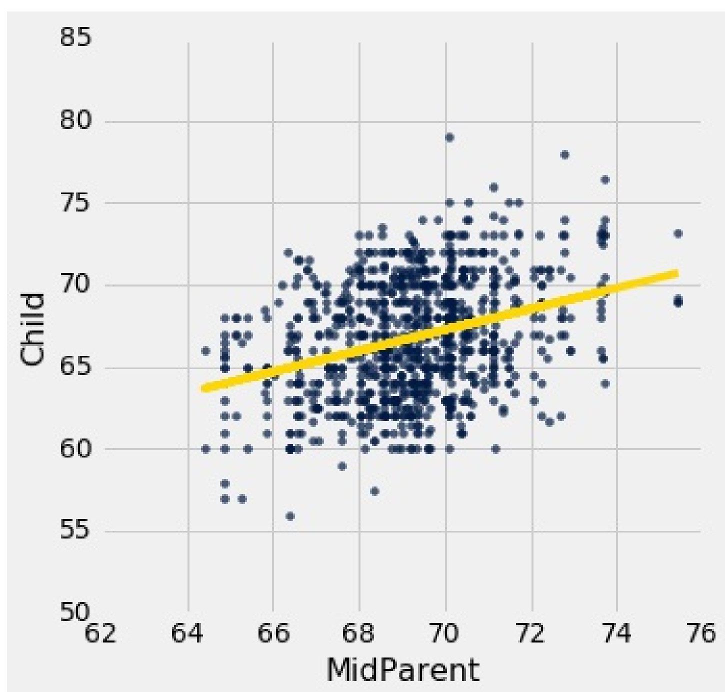


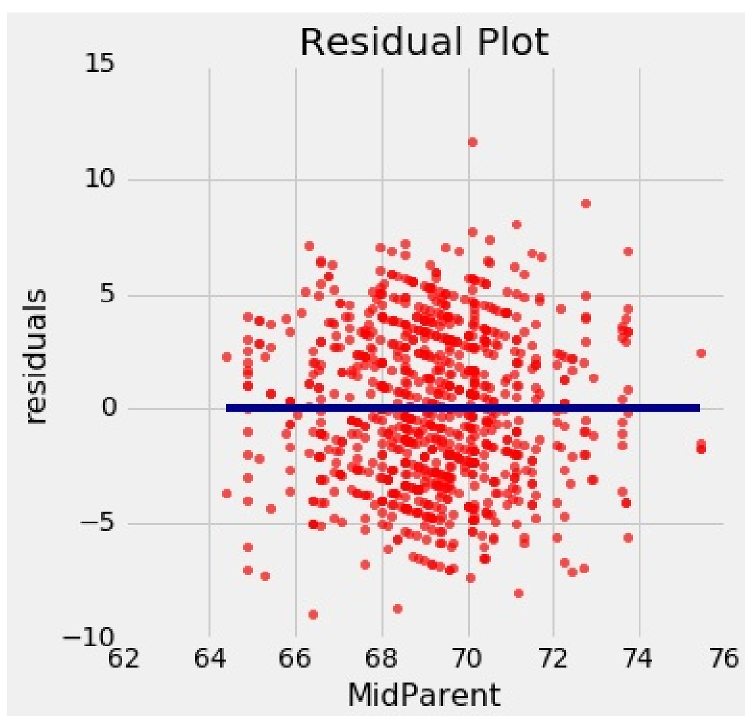
双亲身高在横轴上，就像原始散点图中一样。 但是现在纵轴显示了残差。 请注意，该图看上去以 `y=0` 的横线为中心（以深蓝色显示）。 还要注意，绘图没有显示上升或下降的趋势。 我们稍后会观察到所有的回归都是如此。

回归诊断

残差图有助于我们直观评估线性回归分析的质量。 这种评估被称为诊断。 函数 `regression_diagnostic_plots` 绘制原始散点图以及残差图，以便于比较。

```
def regression_diagnostic_plots(table, x, y):
    scatter_fit(table, x, y)
    residual_plot(table, x, y)
regression_diagnostic_plots(heights, 'MidParent', 'Child')
```





这个残差图表明，线性回归是合理的估计方法。注意残差关于 `y=0` 的横线上下对称分布，相当于原始散点图大致上下对称。还要注意，绘图的垂直延伸，在子女身高最常见的值上相当均匀。换句话说，除了一些离群点之外，绘图并不是一些地方窄。另一些地方宽。

换句话说，在预测变量的观察范围内，回归的准确性似乎是相同的。

良好回归的残差图不显示任何规律。在预测变量的范围内，残差在 `y=0` 的直线处上下相同。

检测非线性

绘制数据的散点图，通常表明了两个变量之间的关系是否是非线性的。 然而，通常情况下，残差图中比原始散点图中更容易发现非线性。 这通常是因为这两个图的规模：残差图允许我们放大错误，从而更容易找出规律。



我们的数据是海牛的年龄和长度的数据集，这是一种海洋哺乳动物（维基共享资源图）。 数据在一个名为 `dugong` 的表中。 年龄以年为单位，长度以米为单位。 因为海牛通常不跟踪他们的生日，年龄是根据他们的牙齿状况等变量来估计的。

```
dugong = Table.read_table('http://www.statsci.org/data/oz/dugongs.txt')
dugong = dugong.move_to_start('Length')
dugong
```

Length	Age
1.8	1
1.85	1.5
1.87	1.5
1.77	1.5
2.02	2.5
2.27	4
2.15	5
2.26	5
2.35	7
2.47	8

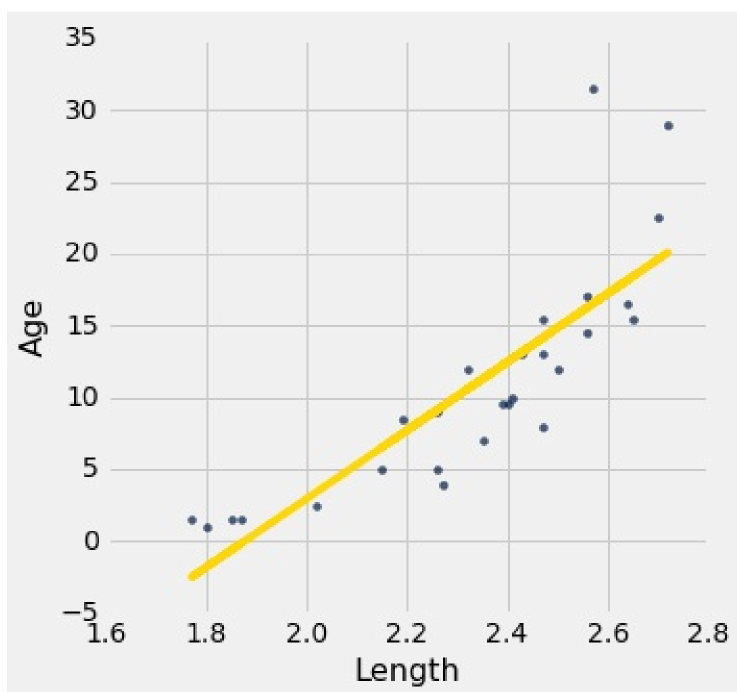
（省略了 17 行）

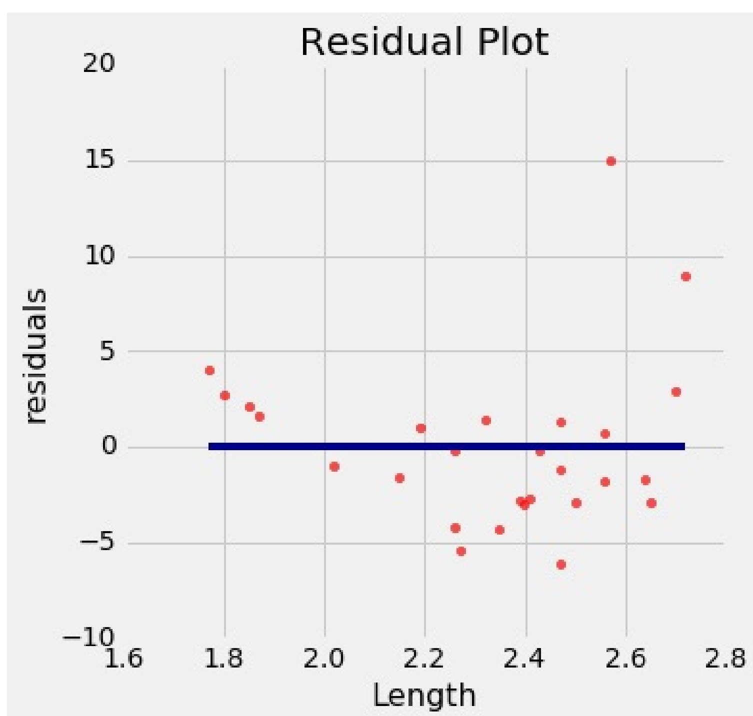
如果我们可以衡量海牛的长度，对于它的年龄我们可以说什么呢？ 让我们来看看我们的数据说了什么。 这是一个长度（预测变量）和年龄（响应变量）的回归。 这两个变量之间的相关性相当大，为 0.83。

```
correlation(dugong, 'Length', 'Age')
0.82964745549057139
```

尽管相关性仍然很高，绘图显示出曲线规律，在残差图中更加明显。

```
regression_diagnostic_plots(dugong, 'Length', 'Age')
```





虽然你可以发现原始散点图中的非线性，但在残差图中更明显。

在长度的较低一端，残差几乎都是正的；然后他们几乎都是负的；然后在较高一端，残差再次为正。 换句话说，回归估计值过高，然后过低，然后过高。 这意味着使用曲线而不是直线来估计年龄会更好。

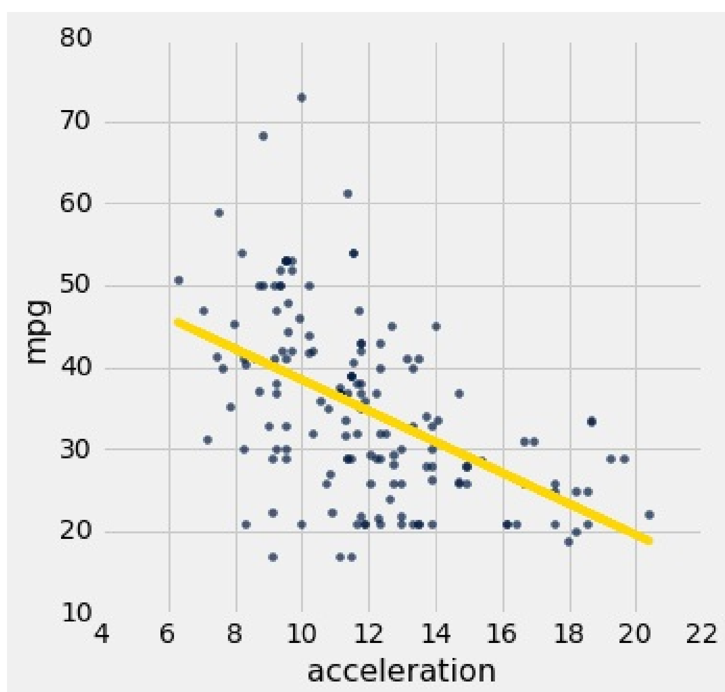
当残差图显示了规律时，变量之间可能存在非线性关系。

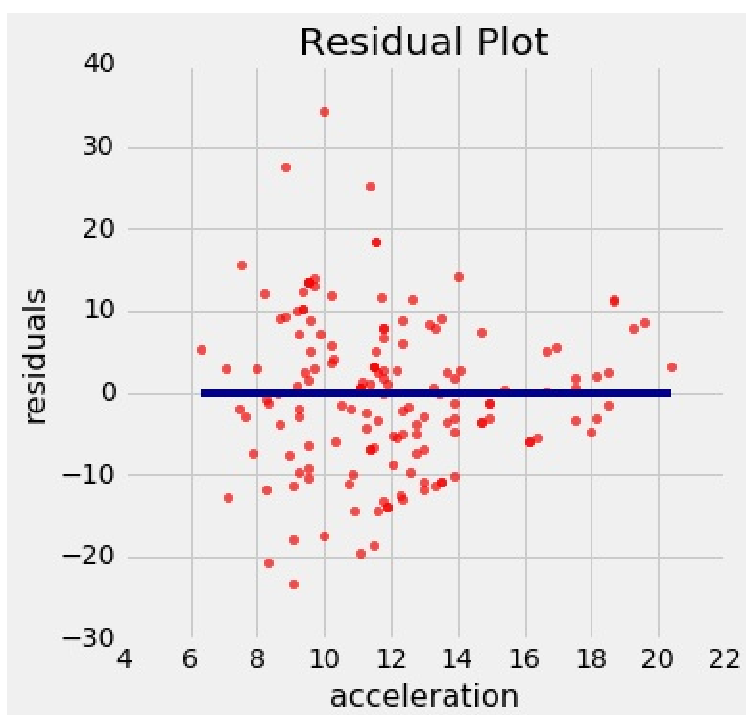
检测异方差

异方差这个词，那些准备拼写游戏的人肯定会感兴趣。 对于数据科学家来说，其兴趣在于它的意义，即“不均匀延伸”。

回想一下 `hybrid` 表，包含美国混合动力汽车的数据。这是燃油效率对加速度的回归。这个关联是负面的：加速度高的汽车往往效率较低。

```
regression_diagnostic_plots(hybrid, 'acceleration', 'mpg')
```





注意残差图在加速度的较低一端变得发散。 换句话说，对于较低的加速度，误差的大小的变化比较高值更大。 残差图中比原始的散点图中更容易注意到不均匀的变化。

如果残差图显示 `y=0` 的横线处的不均匀变化，则在预测变量的范围内，回归的估计不是同等准确的。

数值诊断

除了可视化之外，我们还可以使用残差的数值属性来评估回归的质量。 我们不会在数学上证明这些属性。 相反，我们将通过计算来观察它们，看看它们告诉我们回归的什么东西。

下面列出的所有事实都适用于散点图的所有形状，无论它们是否是线性的。

残差图不展示形状

对于每一个线性回归，无论是好还是坏，残差图都不展示任何趋势。 总的来说，它是平坦的。 换句话说，残差和预测变量是不相关的。

你可以在上面所有的残差图中看到它。 我们还可以计算每种情况下，预测变量和残差之间的相关性。

```
correlation(heights, 'MidParent', 'Residual')
-2.7196898076470642e-16
```

这看起来不是零，但它是个很小的数字，除了由于计算的舍入误差之外，它就是零。 在这里也一样，取小数点后 10 位。 减号是因为上面的舍入。

```
round(correlation(heights, 'MidParent', 'Residual'), 10)
-0.0
dugong = dugong.with_columns(
       'Fitted Value', fit(dugong, 'Length', 'Age'),
       'Residual', residual(dugong, 'Length', 'Age')
)
round(correlation(dugong, 'Length', 'Residual'), 10)
0.0
```

残差的均值

不管散点图的形状如何，剩余的均值都是 0。

这类似于这样一个事实，如果你选取任何数值列表并计算距离均值的偏差的列表，则偏差的均值为 0。

在上面的所有残差图中，你看到 `y=0` 的横线穿过图的中心。 这是这个事实的可视化。

作为一个数值示例，这里是高尔顿数据集中，基于双亲高度的子女高度的回归的残差均值。

```
round(np.mean(heights.column('Residual')), 10)
0.0
```

海牛长度和年龄的回归的残差均值也是一样。 残差均值为 0，除了舍入误差。

```
round(np.mean(dugong.column('Residual')), 10)
0.0
```

残差的标准差

无论散点图的形状如何，残差的标准差是响应变量的标准差的一个比例。 比例是 $\sqrt{1-r^2}$。

$$\text{SD of residuals} ~=~ \sqrt{1 - r^2} \cdot \text{SD of }y$$

我们将很快看到，它如何衡量回归估计的准确性。 但首先，让我们通过例子来确认。

在子女身高和双亲身高的案例中，残差的标准差约为 3.39 英寸。

```
np.std(heights.column('Residual'))
3.3880799163953426
```

这和响应变量的标准差乘`sqrt(1 - r^2)`相同。

```
r = correlation(heights, 'MidParent', 'Child')
np.sqrt(1 - r**2) * np.std(heights.column('Child'))
3.3880799163953421
```

混合动力汽车的加速和里程的回归也是如此。 相关性`r`是负数（约 -0.5），但`r^2`是正数，所以`sqrt(1 - r^2)`是一个分数。

```
r = correlation(hybrid, 'acceleration', 'mpg')
r
-0.5060703843771186
hybrid = hybrid.with_columns(
     'fitted mpg', fit(hybrid, 'acceleration', 'mpg'),
     'residual', residual(hybrid, 'acceleration', 'mpg')
)
np.std(hybrid.column('residual')), np.sqrt(1 - r**2)*np.std(hybrid.column('mpg'))
(9.4327368334302903, 9.4327368334302903)
```

现在让我们看看，残差的标准差是如何衡量回归的好坏。请记住，残差的均值为 0。因此，残差的标准差越小，则残差越接近于 0。换句话说，如果残差的标准差小，那么回归中的总体误差就小。

极端情况是`r = 1`或`r = -1`。在这两种情况下，`sqrt(1 - r^2) = 0`。因此，残差的均值为 0，标准差为 0，因此残差都等于 0。回归线确实是完美的估计。我们在本章的前面看到，如果`r = ± 1`，散点图是一条完美的直线，与回归线相同，所以回归估计中确实没有错误。

但通常`r`不是极端的。如果`r`既不是`±1`也不是 0，那么`sqrt(1 - r^2)`是一个适当的分数，并且回归估计的误差大小，整体上大致在 0 和`y`的标准差之间。

最糟糕的情况是 `r = 0` 。那么 `sqrt(1 - r^2)` = 1，残差的标准差等于 `y` 的标准差。这与观察结果一致，如果 `r = 0` 那么回归线就是 `y` 的均值上的一条横线。在这种情况下，回归的均方根误差是距离 `y` 的平均值的偏差的均方根，这是 `y` 的标准差。实际上，如果 `r = 0` ，那么这两个变量之间就没有线性关联，所以使用线性回归没有任何好处。

另一种解释 `r` 的方式

我们可以重写上面的结果，不管散点图的形状如何：

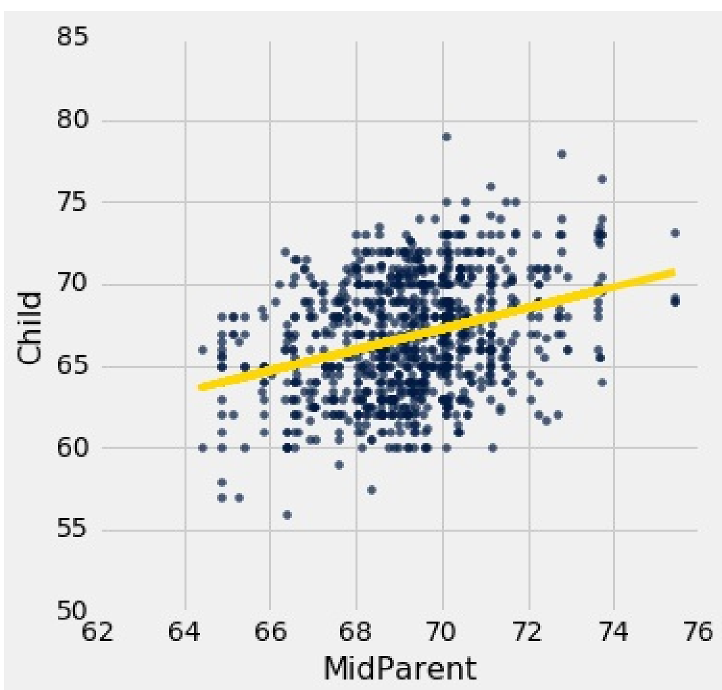
$$\text{SD of residuals} ~=~ \sqrt{1 - r^2} \cdot \text{SD of } y$$

互补的结果是，无论散点图的形状如何，拟合值的标准差是观察值 `y` 的标准差的一个比例。比例是 `|r|` 。

$$\frac{\text{SD of fitted values}}{\text{SD of } y} ~=~ |r|$$

要查看比例在哪里出现，请注意拟合值全部位于回归线上，而 `y` 的观测值是散点图中所有点的高度，并且更加可变。

```
scatter_fit(heights, 'MidParent', 'Child')
```



拟合值的范围在 64 到 71 之间，而所有子女的身高则变化很大，大约在 55 到 80 之间。

为了在数值上验证结果，我们只需要计算双方的一致性。

```
correlation(heights, 'MidParent', 'Child')
0.32094989606395924
```

这里是出生体重的拟合值的标准差与观察值的标准差的比值：

```
np.std(heights.column('Fitted Value'))/np.std(heights.column('Child'))
0.32094989606395957
```

这个比例等于 `r`，证实了我们的结果。

绝对值出现在哪里？首先要注意的是，标准差不能是负数，标准差的比值也不行。那么当 `r` 是负数时会发生什么呢？燃油效率和加速度的例子将向我们展示。

```
correlation(hybrid, 'acceleration', 'mpg')
-0.5060703843771186
np.std(hybrid.column('fitted mpg'))/np.std(hybrid.column('mpg'))
0.5060703843771186
```

两个标准差的比值就是 `|r|`。

解释这个结果的更标准的方法是，回想一下：

$$\mbox{variance} ~=~ \mbox{mean squared deviation from average} ~=~ \mbox{SD}^2$$

因此，对结果的两边取平方：

$$\frac{\mbox{variance of fitted values}}{\mbox{variance of }y} ~=~ r^2$$

十四、回归的推断

> 原文：Inference for Regression
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

到目前为止，我们对变量之间关系的分析纯粹是描述性的。我们知道如何找到穿过散点图的最佳直线来绘制。在所有直线中它的估计的均方误差最小，从这个角度来看，这条线是最好的。

但是，如果我们的数据是更大总体的样本呢？如果我们在样本中发现了两个变量之间的线性关系，那么对于总体也是如此嘛？它会是完全一样的线性关系吗？我们可以预测一个不在我们样本中的新的个体的响应变量吗？

如果我们认为，散点图反映了被绘制的两个变量之间的基本关系，但是并没有完全规定这种关系，那么就会出现这样的推理和预测问题。例如，出生体重与孕期的散点图，显示了我们样本中两个变量之间的精确关系；但是我们可能想知道，对于抽样总体中的所有新生儿或实际中的一般新生儿，这样的关系是否是真实的，或者说几乎是正确的。

一如既往，推断思维起始于仔细检查数据的假设。一组假设被称为模型。大致线性的散点图中的一组随机性的假设称为回归模型。

回归模型

简而言之，这样的模型认为，两个变量之间的底层关系是完全线性的；这条直线是我们想要识别的信号。但是，我们无法清楚地看到这条线。我们看到的是分散在这条线上的点。在每一点上，信号都被随机噪声污染。因此，我们的推断目标是将信号从噪声中分离出来。

更详细地说，回归模型规定了，散点图中的点是随机生成的，如下所示。

- `x` 和 `y` 之间的关系是完全线性的。我们看不到这个“真实直线”，但它是存在的。
- 散点图通过将线上的点垂直移动，或上或下来创建，如下所示：
- 对于每个 `x`，找到真实直线上的相应点（即信号），然后生成噪声或误差。
- 误差从误差总体中带放回随机抽取，总体是均值为 0 的正态分布。
- 创建一个点，横坐标为 `x`，纵坐标为“`x` 处的真实高度加上误差”。
- 最后，从散点图中删除真正的线，只显示创建的点。

基于这个散点图，我们应该如何估计真实直线？ 我们可以使其穿过散点图的最佳直线是回归线。 所以回归线是真实直线的自然估计。

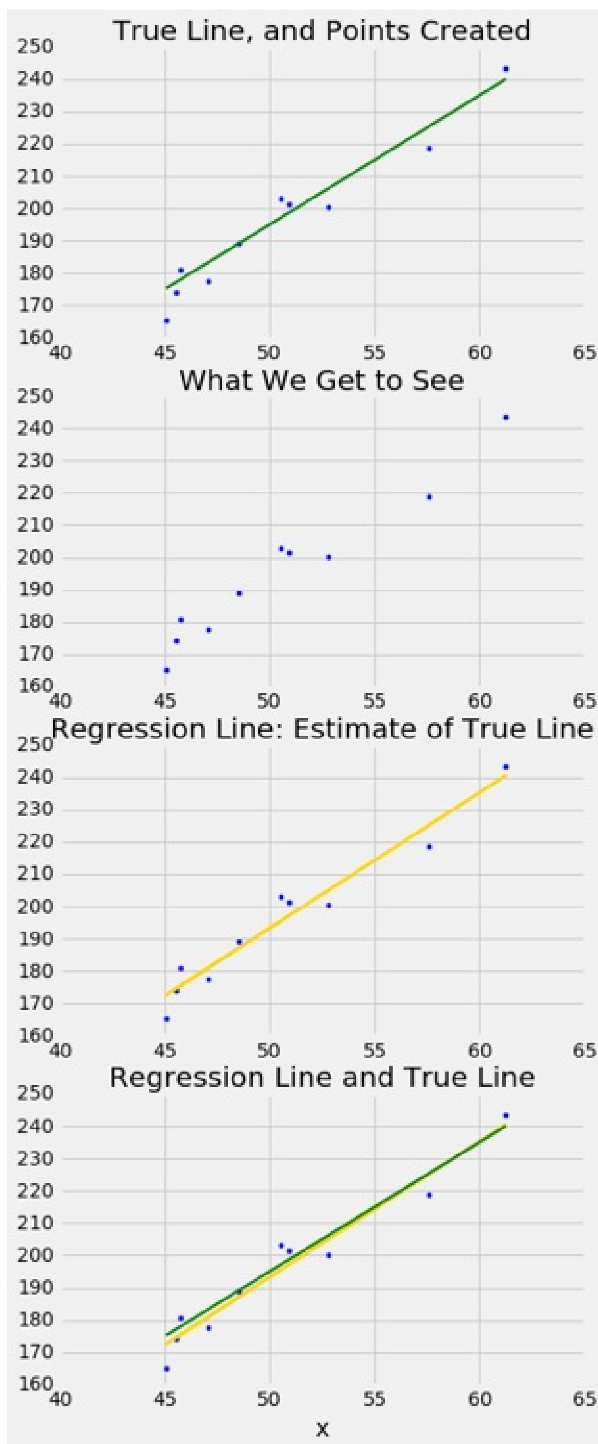
下面的模拟显示了回归直线与真实直线的距离。 第一个面板显示如何从真实直线生成散点图。 第二个显示我们看到的散点图。 第三个显示穿过散点图的回归线。 第四个显示回归线和真实直线。

为了运行模拟，请使用三个参数调用`draw_and_compare`函数：真实直线的斜率，真实直线的截距以及样本量。

运行模拟几次，用不同的斜率和截距，以及不同的样本量。 因为所有的点都是根据模型生成的，所以如果样本量适中，你会看到回归线是真实直线的一个良好估计。

```
# The true line,
# the points created,
# and our estimate of the true line.
# Arguments: true slope, true intercept, number of points

draw_and_compare(4, -5, 10)
```



实际上，我们当然不会看到真实直线。 模拟结果表明，如果回归模型看起来合理，并且如果我们拥有大型样本，那么回归线就是真实直线的一个良好近似。

真实斜率的推断

我们的模拟表明，如果回归模型成立，并且样本量很大，则回归线很可能接近真实直线。 这使我们能够估计真实直线的斜率。

我们将使用我们熟悉的母亲和她们的新生儿的样本，来开发估计真实直线的斜率的方法。 首先，我们来看看我们是否相信，回归模型是一系列适当假设，用于描述出生体重和孕期之间的关系。

```
correlation(baby, 'Gestational Days', 'Birth Weight')
0.40754279338885108
```

总的来说，散点图看起来相当均匀地分布在这条线上，尽管一些点分布在主云形的周围。 相关系数为 0.4，回归线斜率为正。

这是否反映真实直线斜率为正的事实？ 为了回答这个问题，让我们看看我们能否估计真实斜率。 我们当然有了一个估计：我们的回归线斜率。 这大约是 0.47 盎司每天。

```
slope(baby, 'Gestational Days', 'Birth Weight')
0.46655687694921522
```

但是如果散点图出现的方式不同，回归线会有所不同，可能会有不同的斜率。 我们如何计算，斜率可能有多么不同？

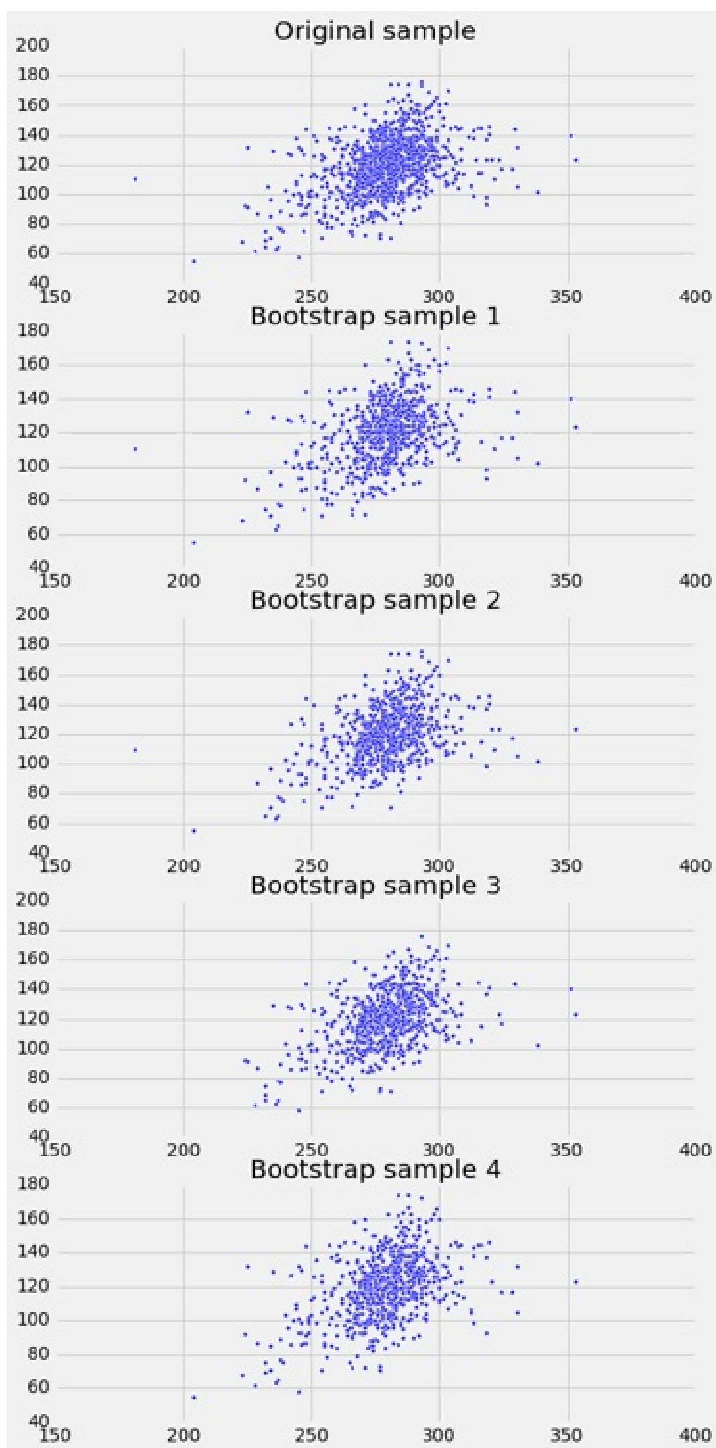
我们需要点的另一个样本，以便我们可以绘制回归线穿过新的散点图，并找出其斜率。 但另一个样本从哪里得到呢？

你猜对了 - 我们将自举我们的原始样本。 这会给我们自举的散点图，通过它我们可以绘制回归线。

自举散点图

我们可以通过对原始样本带放回地随机抽样，来模拟新样本，它的次数与原始样本量相同。 这些新样本中的每一个都会给我们一个散点图。 我们将这个称为自举散点图，简而言之，我们将调用整个过程来自举散点图。

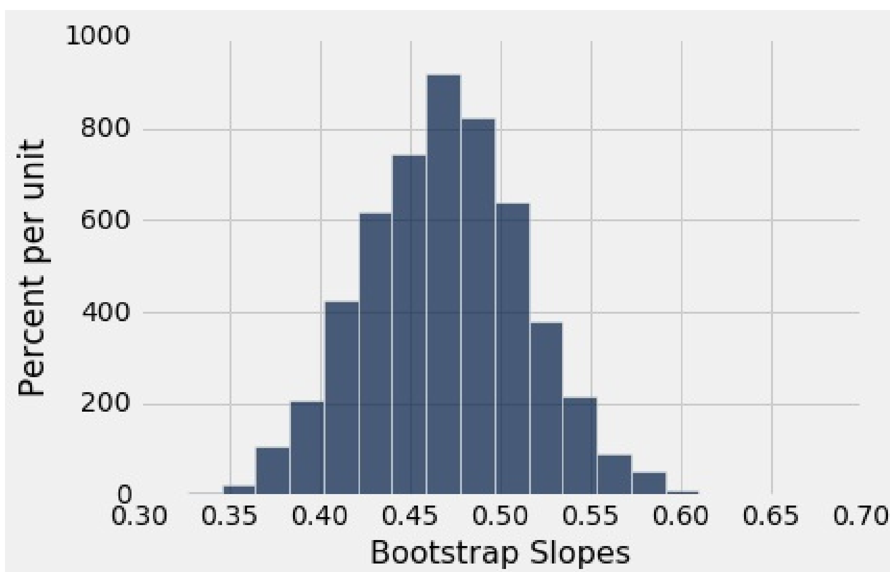
这里是来自样本的原始散点图，以及自举重采样过程的四个复制品。 请注意，重采样散点图通常比原始图稀疏一点。 这是因为一些原始的点没有在样本中被选中。



估计真实斜率

我们可以多次自举散点图，并绘制穿过每个自举图的回归线。 每条线都有一个斜率。 我们可以简单收集所有的斜率并绘制经验直方图。 回想一下，在默认情况下， `sample` 方法带放回地随机抽取，次数与表中的行数相同。 也就是说， `sample` 默认生成一个自举样本。

```
slopes = make_array()
for i in np.arange(5000):
    bootstrap_sample = baby.sample()
    bootstrap_slope = slope(bootstrap_sample, 'Gestational Days', 'Birth Weight')
    slopes = np.append(slopes, bootstrap_slope)
Table().with_column('Bootstrap Slopes', slopes).hist(bins=20)
```



然后，我们可以使用 `percentile` 方法，为真实直线的斜率构建约 95% 置信区间。 置信区间从 5000 个自举斜率的第 2.5 百分位数，延伸到第 97.5 百分位数。

```
left = percentile(2.5, slopes)
right = percentile(97.5, slopes)
left, right
(0.38209399211893086, 0.56014757838023777)
```

用于自举斜率的函数

让我们收集我们估计斜率的方法的所有步骤，并定义函数 `bootstrap_slope` 来执行它们。 它的参数是表的名称，预测变量和响应变量的标签，以及自举复制品的所需数量。 在每个复制品中，该函数自举原始散点图并计算所得回归线的斜率。 然后绘制所有生成的斜率的直方图，并打印由斜率的“中间 95%”组成的区间。

```
def bootstrap_slope(table, x, y, repetitions):

    # For each repetition:
    # Bootstrap the scatter, get the slope of the regression line,
    # augment the list of generated slopes
    slopes = make_array()
    for i in np.arange(repetitions):
        bootstrap_sample = table.sample()
        bootstrap_slope = slope(bootstrap_sample, x, y)
        slopes = np.append(slopes, bootstrap_slope)

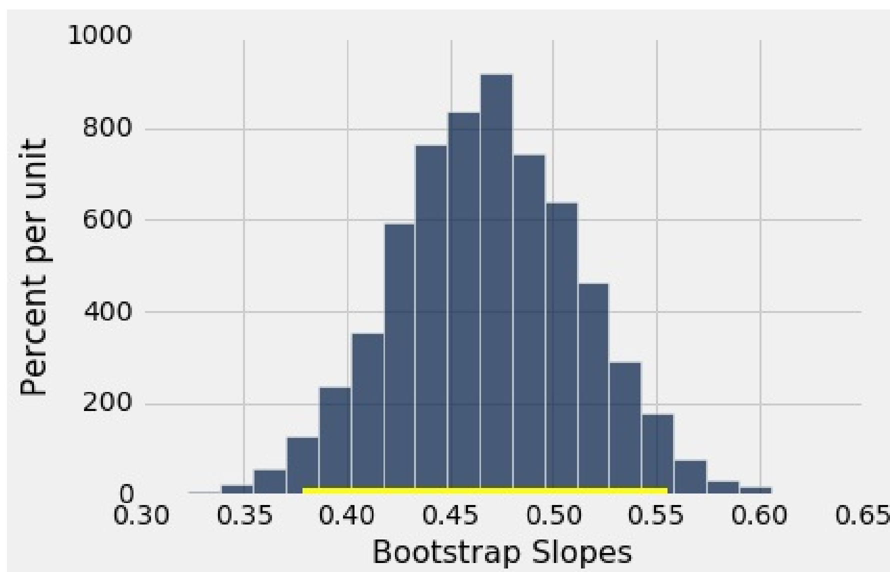
    # Find the endpoints of the 95% confidence interval for the true slope
    left = percentile(2.5, slopes)
    right = percentile(97.5, slopes)

    # Slope of the regression line from the original sample
    observed_slope = slope(table, x, y)

    # Display results
    Table().with_column('Bootstrap Slopes', slopes).hist(bins=20)
    plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=8);
    print('Slope of regression line:', observed_slope)
    print('Approximate 95%-confidence interval for the true slope:')
    print(left, right)
```

当响应变量为出生体重，预测变量为孕期时，我们调用 `bootstrap_slope` 来找到真实斜率的置信区间，我们得到了一个区间，非常接近我们之前获得的东西：大约 0.38 到 0.56 盎司每天。

```
bootstrap_slope(baby, 'Gestational Days', 'Birth Weight', 5000)
Slope of regression line: 0.466556876949
Approximate 95%-confidence interval for the true slope:
0.378663152966 0.555005146304
```

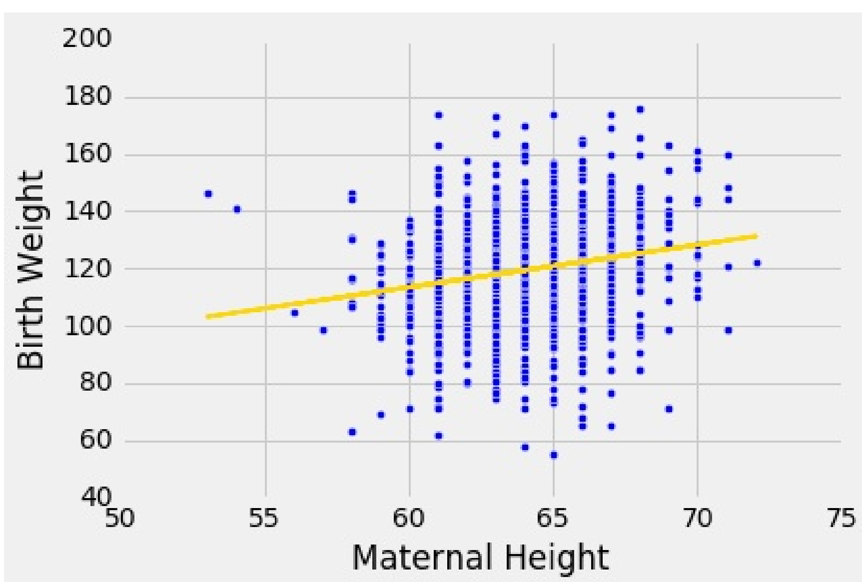


现在我们有一个函数，可以自动完成估计在回归模型中展示斜率的过程，我们也可以在其他变量上使用它。

例如，我们来看看出生体重与母亲身高的关系。 更高的女性往往有更重的婴儿吗？

回归模型似乎是合理的，基于散点图，但相关性不高。 这只有大约 0.2。

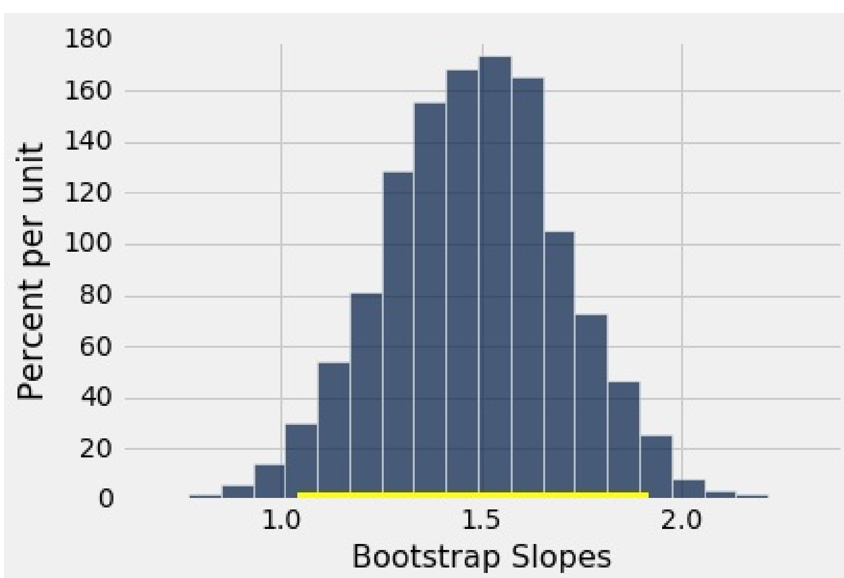
```
scatter_fit(baby, 'Maternal Height', 'Birth Weight')
```




```
correlation(baby, 'Maternal Height', 'Birth Weight')
0.20370417718968034
```

像之前一样，我们使用 `bootstrap_slope` 来估计回归模型中真实直线的斜率。

```
bootstrap_slope(baby, 'Maternal Height', 'Birth Weight', 5000)
Slope of regression line: 1.47801935193
Approximate 95%-confidence interval for the true slope:
1.0403083964 1.91576886223
```



真实斜率的 95% 置信区间，从约 1 延伸到约 1.9 盎司每英寸。

真实斜率可能为 0 嘛？

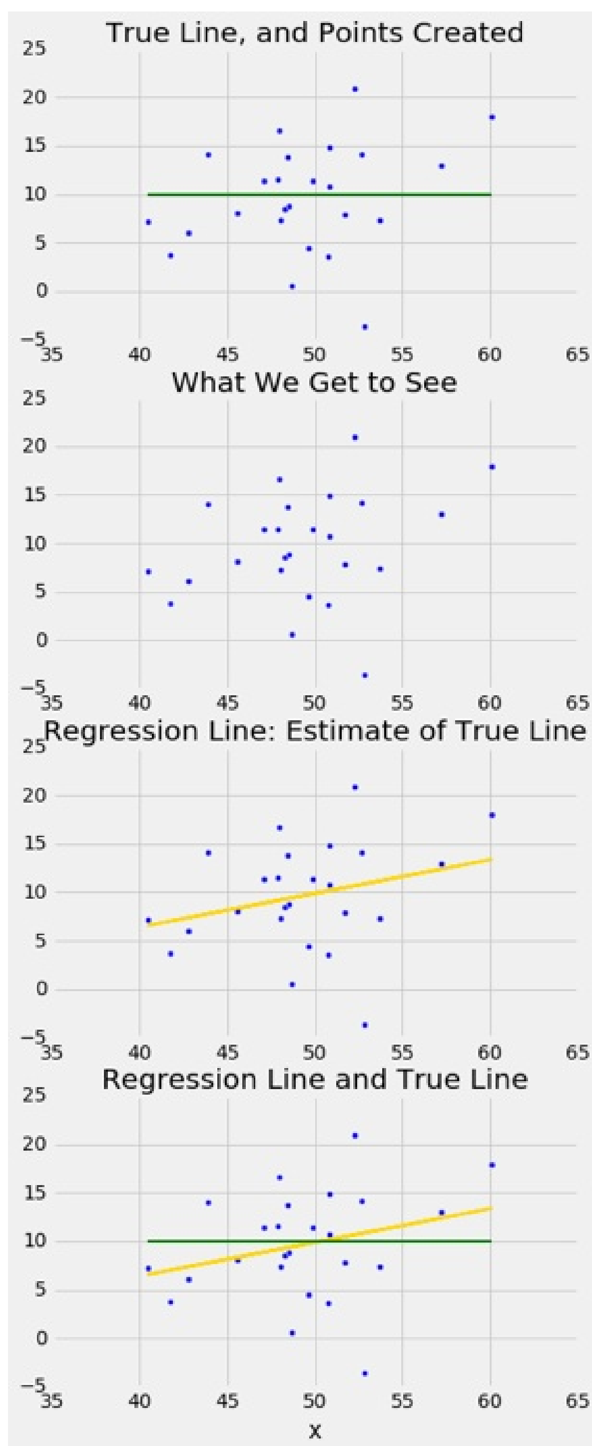
假设我们相信我们的数据遵循回归模型，并且我们拟合回归线来估计真实直线。 如果回归线不完全是平的，几乎总是如此，我们将观察到散点图中的一些线性关联。

但是，如果这种观察是假的呢？ 换句话说，如果真实直线是平的 - 也就是说，这两个变量之间没有线性关系 - 我们观察到的联系，只是由于从样本中产生点的随机性。

这是一个模拟，说明了为什么会出现这个问题。 我们将再次调用`draw_and_compare`函数，这次要求真实斜率为 0。我们的目标是，观察我们的回归线是否显示不为 0 的斜率。

请记住函数`draw_and_compare`的参数是真实直线的斜率和截距，以及要生成的点的数量。

```
draw_and_compare(0, 10, 25)
```



运行模拟几次，每次保持真实直线的斜率为 0 。你会注意到，虽然真实直线的斜率为 0，但回归线的斜率通常不为 0。回归线有时会向上倾斜，有时会向下倾斜，每次都给我们错误的印象，即这两个变量是相关的。

为了确定我们所看到的斜率是否真实，我们想测试以下假设：

原假设。真实直线的斜率是 0。

备选假设。真实直线的斜率不是 0。

我们很有条件来实现它。由于我们可以为真实斜率构建一个 95% 的置信区间，我们所要做的就是看区间是否包含 0。

如果没有，那么我们可以拒绝原假设（P 值为 5% 的截断值）。

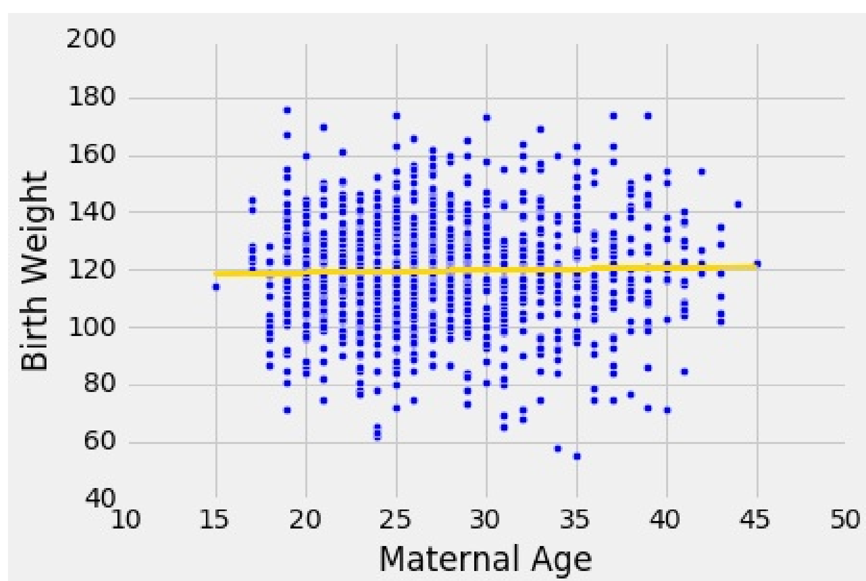
如果真实斜率的置信区间确实包含 0，那么我们没有足够的证据来拒绝原假设。也许我们看到的斜率是假的。

我们在一个例子中使用这个方法。假设我们试图根据母亲的年龄来估计新生儿的出生体重。根据样本，根据母亲年龄估计出生体重的回归线的斜率为正，约为 0.08 盎司每年。

```
slope(baby, 'Maternal Age', 'Birth Weight')
0.085007669415825132
```

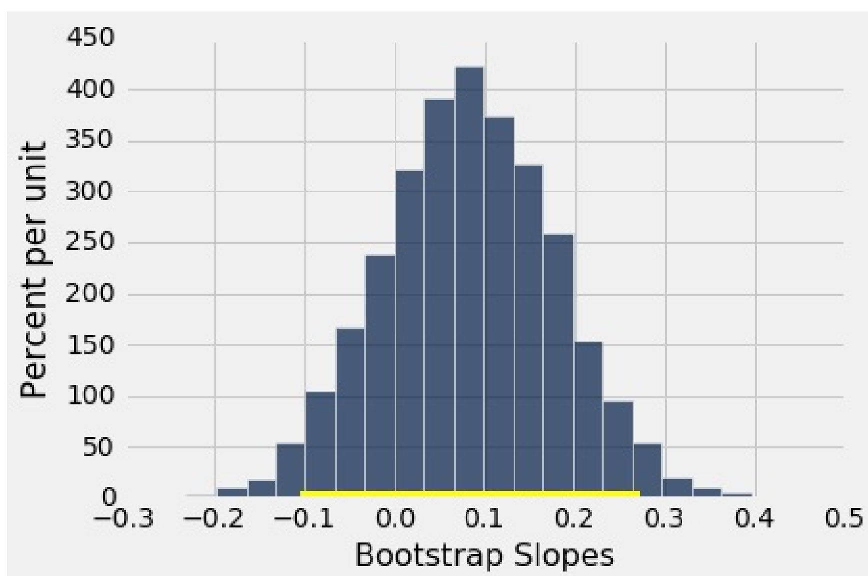
虽然斜率为正，但是很小。 回归线非常接近平的，这就产生了一个问题，真实直线是否是平的。

```
scatter_fit(baby, 'Maternal Age', 'Birth Weight')
```



我们可以使用 `bootstrap_slope` 来估计真实直线的斜率。 计算表明，真实斜率的约 95% 的自举置信区间左端为负，右端为正 - 换句话说，区间包含 0。

```
bootstrap_slope(baby, 'Maternal Age', 'Birth Weight', 5000)
Slope of regression line: 0.0850076694158
Approximate 95%-confidence interval for the true slope:
-0.104335243815 0.272791852339
```



因为区间包含 0，所以我们不能拒绝原假设，母亲年龄与新生儿出生体重之间的真实线性关系的斜率为 0。基于此分析，使用母亲年龄作为预测变量，基于回归模型预测出生体重是不明智的。

预测区间

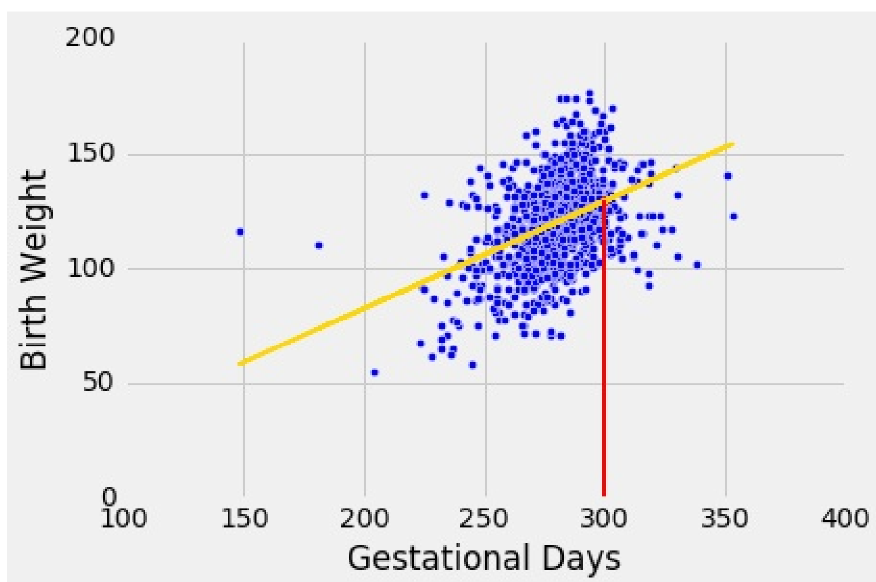
回归的主要用途之一是对新个体进行预测，这个个体不是我们原始样本的一部分，但是与样本个体相似。在模型的语言中，我们想要估计新值 `x` 的 `y` 。

我们的估计是真实直线在 `x` 处的高度。当然，我们不知道真实直线。我们使用我们的样本点的回归线来代替。

给定值 `x` 的拟合值，是基于 `x` 值的 `y` 的回归估计。换句话说，给定值 `x` 的拟合值就是回归线在 `x` 处的高度。

假设我们试图根据孕期天数来预测新生儿的出生体重。我们在前面的章节中看到，这些数据非常适合回归模型，真实直线的斜率的 95% 置信区间不包含 0。因此，我们的预测似乎是合理的。

下图显示了预测位于回归线上的位置。红线是 `x = 300` 。



红线与回归线的相交点的高度是孕期天数 300 的拟合值。

函数 `fitted_value` 计算这个高度。像函数的相关性，斜率和截距一样，它的参数是表的名称和 `x` 和 `y` 的列标签。但是它也需要第四个参数，即 `x` 的值，在这个值上进行估算。

```
def fitted_value(table, x, y, given_x):
    a = slope(table, x, y)
    b = intercept(table, x, y)
    return a * given_x  + b
```

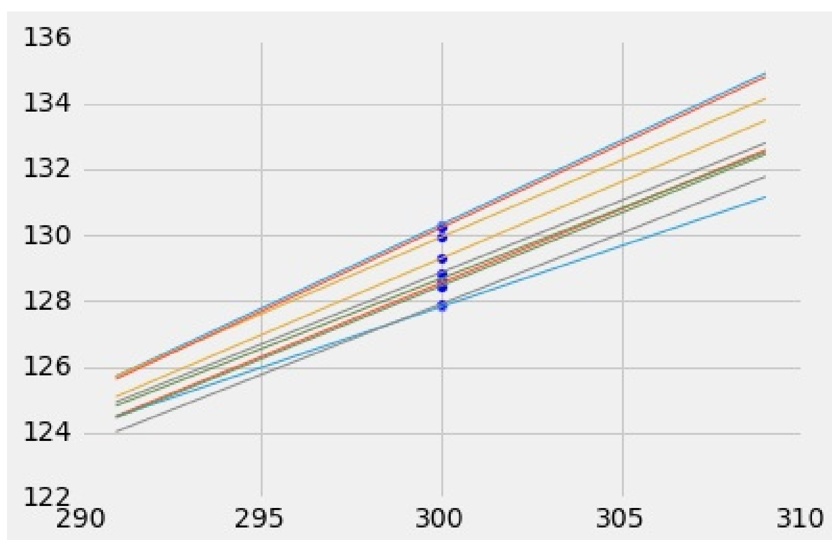
孕期天数 300 的拟合值约为 129.2 盎司。 换句话说，对于孕期为 300 天的孕妇，我们估计的新生儿体重约为 129.2 盎司。

```
fit_300 = fitted_value(baby, 'Gestational Days', 'Birth Weight', 300)
fit_300
129.2129241703143
```

预测的可变性

我们已经开发了一种方法，使用我们样本中的数据，根据孕期天数预测新生儿的体重。 但作为数据科学家，我们知道样本可能有所不同。 如果样本不同，回归线也会不一样，我们的预测也是。 为了看看我们的预测有多好，我们必须了解预测的可变性。

为此，我们必须生成新的样本。 我们可以像上一节那样，通过自举散点图来实现。 然后，我们为每个散点图的复制品拟合回归线，并根据每一行进行预测。 下图显示了 10 条这样的线，以及孕期天数 300 对应的出生体重预测。




```
lines
```

slope	intercept	prediction at x=300
0.503931	-21.6998	129.479
0.53227	-29.5647	130.116
0.518771	-25.363	130.268
0.430556	-1.06812	128.099
0.470229	-11.7611	129.308
0.48713	-16.5314	129.608
0.51241	-23.2954	130.428
0.52473	-27.2053	130.214
0.409943	5.22652	128.21
0.468065	-11.6967	128.723

每一行的预测都不相同。 下表显示了 10 条线的斜率、截距以及预测。

自举预测区间

如果我们增加重采样过程的重复次数，我们可以生成预测的经验直方图。这将允许我们创建预测区间，使用为斜率创建自举置信区间时的相同的百分比方法。

让我们定义一个名为 `bootstrap_prediction` 的函数来实现。该函数有五个参数：

- 表的名称
- 预测变量和响应变量的列标签
- 用于预测的 `x` 的值
- 所需的自举重复次数

在每次重复中，函数将自举原始散点图，并基于 `x` 的指定值查找 `y` 的预测值。具体来说，它调用我们在本节前面定义的函数 `fitted_value`，来寻找指定 `x` 处的拟合值。

最后，绘制所有预测值的经验直方图，并打印由预测值的“中间 95%”组成的区间。它还打印基于穿过原始散点图的回归线的预测值。

```
# Bootstrap prediction of variable y at new_x
# Data contained in table; prediction by regression of y based on x
# repetitions = number of bootstrap replications of the original scatter plot

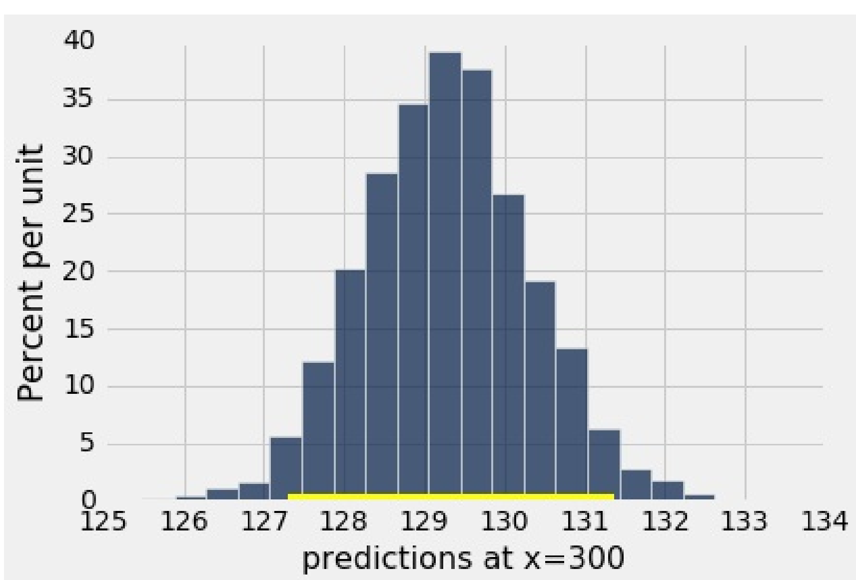
def bootstrap_prediction(table, x, y, new_x, repetitions):

    # For each repetition:
    # Bootstrap the scatter;
    # get the regression prediction at new_x;
    # augment the predictions list
    predictions = make_array()
    for i in np.arange(repetitions):
        bootstrap_sample = table.sample()
        bootstrap_prediction = fitted_value(bootstrap_sample, x, y, new_x)
        predictions = np.append(predictions, bootstrap_prediction)

    # Find the ends of the approximate 95% prediction interval
    left = percentile(2.5, predictions)
    right = percentile(97.5, predictions)

    # Prediction based on original sample
    original = fitted_value(table, x, y, new_x)

    # Display results
    Table().with_column('Prediction', predictions).hist(bins=20)
    plots.xlabel('predictions at x='+str(new_x))
    plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=8);
    print('Height of regression line at x='+str(new_x)+':', original)
    print('Approximate 95%-confidence interval:')
    print(left, right)
bootstrap_prediction(baby, 'Gestational Days', 'Birth Weight', 300, 5000)
Height of regression line at x=300: 129.21292417
Approximate 95%-confidence interval:
127.300774171 131.361729528
```



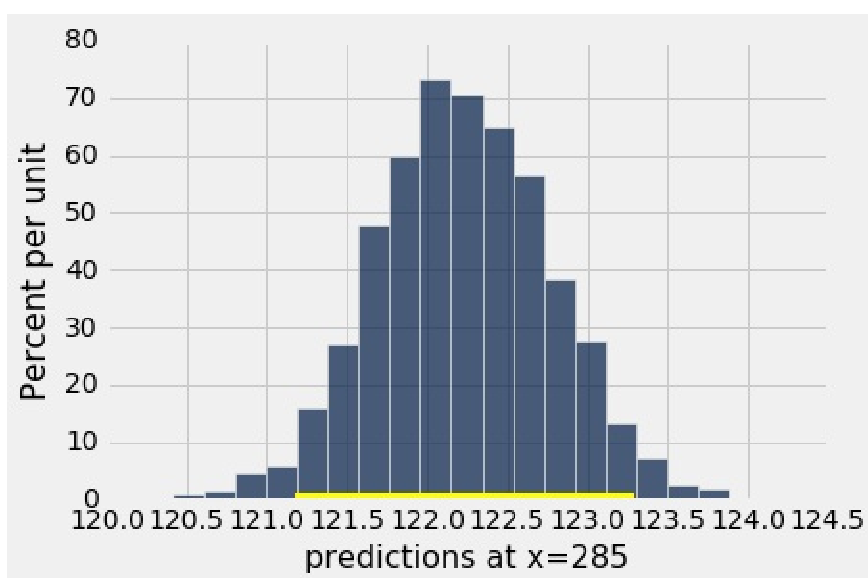
上图显示了基于 5000 次重复的自举过程，孕期天数 300 的预测出生体重的自举经验直方图。经验分布大致是正泰的。

我们已经通过预测的“中间 95%”，即预测的第 2.5 百分位数到第 97.5 百分位数的区间，构建了分数的约 95% 的预测区间。 区间范围从大约 127 到大约 131。基于原始样本的预测是大约 129，接近区间的中心。

改变预测变量的值的效果

下图显示了孕期天数 285 的 5,000 次自举预测的直方图。 基于原始样本的预测是约 122 盎司，区间范围从约 121 盎司到约 123 盎司。

```
bootstrap_prediction(baby, 'Gestational Days', 'Birth Weight', 285, 5000)
Height of regression line at x=285: 122.214571016
Approximate 95%-confidence interval:
121.177089926 123.291373304
```



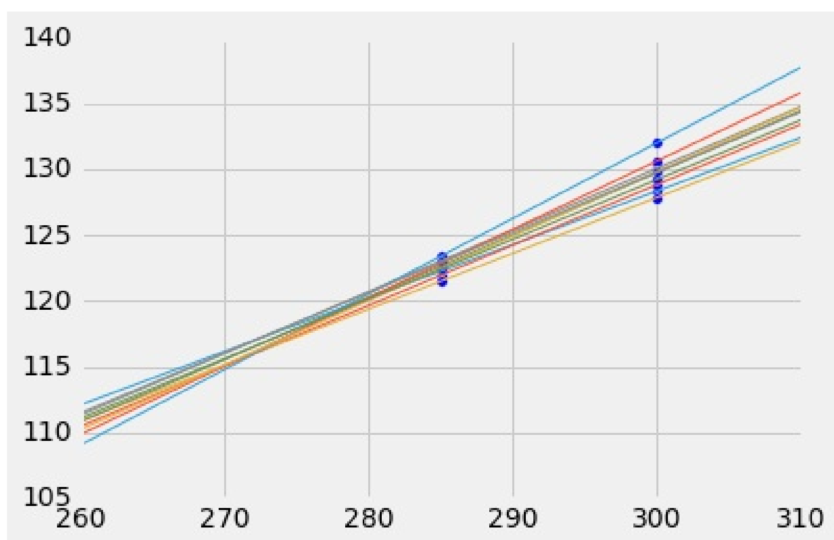
请注意，这个区间比孕妇天数 300 的预测区间更窄。 让我们来调查其原因。

孕妇天数均值约为 279 天：

```
np.mean(baby.column('Gestational Days'))
279.10136286201021
```

所以 285 比 300 更接近分布的中心。 通常，基于自举样本的回归线，在预测变量的分布中心附近彼此更接近。 因此，所有的预测值也更接近。 这解释了预测区间的宽度更窄。

你可以在下面的图中看到这一点，它显示了 10 个自举复制品中每一个的 `x = 285` 和 `x = 300` 的预测值。 通常情况下，直线在 `x = 300` 处比 `x = 285` 处相距更远，因此 `x = 300` 的预测更加可变。



注意事项

我们在本章中进行的所有预测和测试，都假设回归模型是成立的。 具体来说，这些方法假设，散点图中的点由直线上的点产生，然后通过添加随机正态噪声将它们推离直线。

如果散点图看起来不像那样，那么模型可能不适用于数据。 如果模型不成立，那么假设模型为真的计算是无效的。

因此，在开始基于模型进行预测，或者对模型参数进行假设检验之前，我们首先要确定回归模型是否适用于我们的数据。 一个简单的方法就是，按照我们在本节所做的操作，即绘制两个变量的散点图，看看它看起来是否大致线性，并均匀分布在一条线上。 我们还应该使用残差图，执行我们在前一节中开发的诊断。

十五、分类

> 原文：Classification
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

David Wagner 是这一章的主要作者。

机器学习是一类技术，用于自动寻找数据中的规律，并使用它来推断或预测。你已经看到了线性回归，这是一种机器学习技术。本章介绍一个新的技术：分类。

分类就是学习如何根据过去的例子做出预测。我们举了一些例子，告诉我们什么是正确的预测，我们希望从这些例子中学习，如何较好地预测未来。以下是在实践中分类的一些应用领域：

- 他们有一些每个订单的信息（例如，它的总值，订单是否被运送到这个客户以前使用过的地址，是否与信用卡持有人的账单地址相同）。他们有很多过去的订单数据，他们知道哪些过去的订单是欺诈性的，哪些不是。他们想要学习规律，这将帮助他们预测新订单到达时，这些新订单是否有欺诈行为。
- 在线约会网站希望预测：这两个人合适吗？他们有很多数据，他们过去向顾客推荐一些东西，它们就知道了哪个是成功的。当新客户注册时，他们想预测谁可能是他们的最佳伴侣。
- 医生想知道：这个病人是否患有癌症？根据一些实验室测试的结果，他们希望能够预测特定患者是否患有癌症。基于一些实验室测试的测量结果，以及他们是否最终发展成癌症，并且由此他们希望尝试推断，哪些测量结果倾向于癌症（或非癌症）特征，以便能够准确地诊断未来的患者。
- 政客们想预测：你打算为他们投票吗？这将帮助他们将筹款工作集中在可能支持他们的人身上，并将动员工作集中在投票给他们的人身上。公共数据库和商业数据库有大多数人的大量信息，例如，他们是否拥有房屋或房租；他们是否住在富裕的社区还是贫穷的社区；他们的兴趣和爱好；他们的购物习惯；等等。政治团体已经调查了一些选民，并找到了他们计划投票的人，所以他们有一些正确答案已知的例子。

所有这些都是分类任务。请注意，在每个例子中，预测是一个是与否的问题 - 我们称之为二元分类，因为只有两个可能的预测。

在分类任务中，我们想要进行预测的每个个体或情况都称为观测值。我们通常有很多观测值。每个观测值具有多个已知属性（例如，亚马逊订单的总值，或者选民的年薪）。另外，每个观测值都有一个类别，这是对我们关心的问题（例如欺骗与否，或者是否投票）的回答。

当亚马逊预测订单是否具有欺诈性时，每个订单都对应一个单独的观测值。每个观测值都有几个属性：订单的总值，订单是否被运送到此客户以前使用的地址等等。观测值类别为 0 或 1，其中 0 意味着订单不是欺诈，1 意味着订单是欺诈性的。当一个客户生成新的订单时，我们并没有观察到这个订单是否具有欺诈性，但是我们确实观察了这个订单的属性，并且我们会尝试用这些属性来预测它的类别。

分类需要数据。它涉及到发现规律，并且为了发现规律，你需要数据。这就是数据科学的来源。特别是，我们假设我们可以获得训练数据：一系列的观测数据，我们知道每个观测值的类别。这些预分类的观测值集合也被称为训练集。分类算法需要分析训练集，然后提出一个分类器：用于预测未来观测值类别的算法。

分类器不需要是完全有用的。即使准确度低于 100%，它们也可以是有用的。例如，如果在线约会网站偶尔会提出不好的建议，那没关系；他们的顾客已经预期，在他们找到真爱之前需要遇见许多人。当然，你不希望分类器犯太多的错误，但是不必每次都得到正确的答案。

最近邻

在本节中，我们将开发最近邻分类方法。 如果一些代码神秘，不要担心，现在只要把注意力思路上。 在本章的后面，我们将看到如何将我们的想法组织成执行分类的代码。

慢性肾病

我们来浏览一个例子。 我们将使用收集的数据集来帮助医生诊断慢性肾病（CKD）。 数据集中的每一行都代表单个患者，过去接受过治疗并且诊断已知。 对于每个患者，我们都有一组血液测试的测量结果。 我们希望找到哪些测量结果对诊断慢性肾病最有用，并根据他们的血液检查结果，开发一种方法，将未来的患者分类为“CKD”或“非 CKD”。

```
ckd = Table.read_table('ckd.csv').relabeled('Blood Glucose Random', 'Glucose')
ckd
```

Age	Blood Pressure	Specific Gravity	Albumin	Sugar	Red Blood Cells	Pus Cell	Pus C clum
48	70	1.005	4	0	normal	abnormal	preser
53	90	1.02	2	0	abnormal	abnormal	preser
63	70	1.01	3	0	abnormal	abnormal	preser
68	80	1.01	3	2	normal	abnormal	preser
61	80	1.015	2	0	abnormal	abnormal	notpre
48	80	1.025	4	0	normal	abnormal	notpre
69	70	1.01	3	4	normal	abnormal	notpre
73	70	1.005	0	0	normal	normal	notpre
73	80	1.02	2	0	abnormal	abnormal	notpre
46	60	1.01	1	0	normal	normal	notpre

（省略了 148 行）

一些变量是类别（像“异常”这样的词），还有一些是定量的。 定量变量都有不同的规模。 我们将要通过眼睛比较和估计距离，所以我们只选择一些变量并在标准单位下工作。 之后我们就不用担心每个变量的规模。

```
ckd = Table().with_columns(
    'Hemoglobin', standard_units(ckd.column('Hemoglobin')),
    'Glucose', standard_units(ckd.column('Glucose')),
    'White Blood Cell Count', standard_units(ckd.column('White Blood Cell Count')),
    'Class', ckd.column('Class')
)
ckd
```

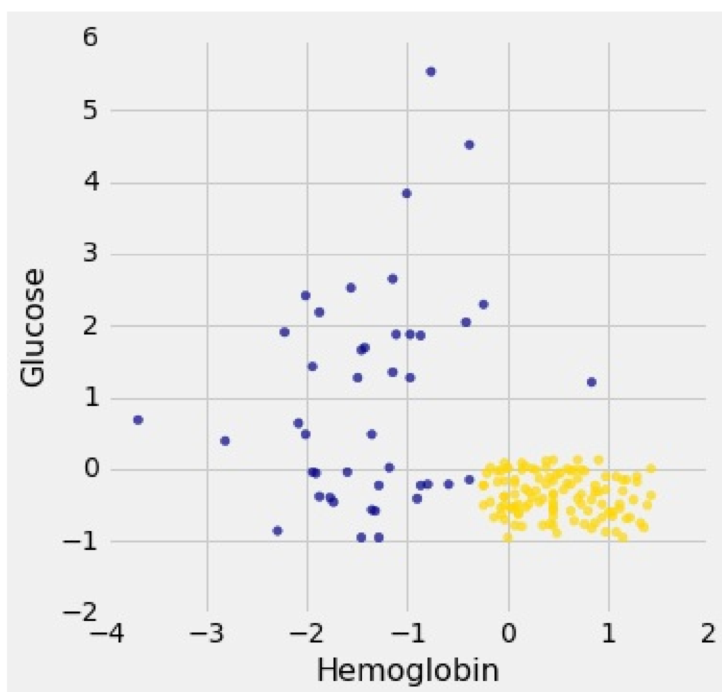
Hemoglobin	Glucose	White Blood Cell Count	Class
-0.865744	-0.221549	-0.569768	1
-1.45745	-0.947597	1.16268	1
-1.00497	3.84123	-1.27558	1
-2.81488	0.396364	0.809777	1
-2.08395	0.643529	0.232293	1
-1.35303	-0.561402	-0.505603	1
-0.413266	2.04928	0.360623	1
-1.28342	-0.947597	3.34429	1
-1.10939	1.87936	-0.409356	1
-1.35303	0.489051	1.96475	1

（省略了 148 行）

我们来看两列，（病人的血液中）血红蛋白水平和血糖水平（一天中的随机时间；没有专门为血液测试禁食）。

我们将绘制一个散点图来显示两个变量之间的关系。 蓝点是 CKD 患者; 金点是非 CKD 的患者。 什么样的医学检验结果似乎表明了 CKD？

```
color_table = Table().with_columns(
    'Class', make_array(1, 0),
    'Color', make_array('darkblue', 'gold')
)
ckd = ckd.join('Class', color_table)
ckd.scatter('Hemoglobin', 'Glucose', colors='Color')
```



假设爱丽丝是不在数据集中的新患者。 如果我告诉你爱丽丝的血红蛋白水平和血糖水平，你可以预测她是否有 CKD 嘛？ 确实看起来可以！ 你可以在这里看到非常清晰的规律：右下角的点代表没有 CKD 的人，其余的倾向于有 CKD 的人。 对于人来说，规律是显而易见的。 但是，我们如何为计算机编程来自动检测这种规律？

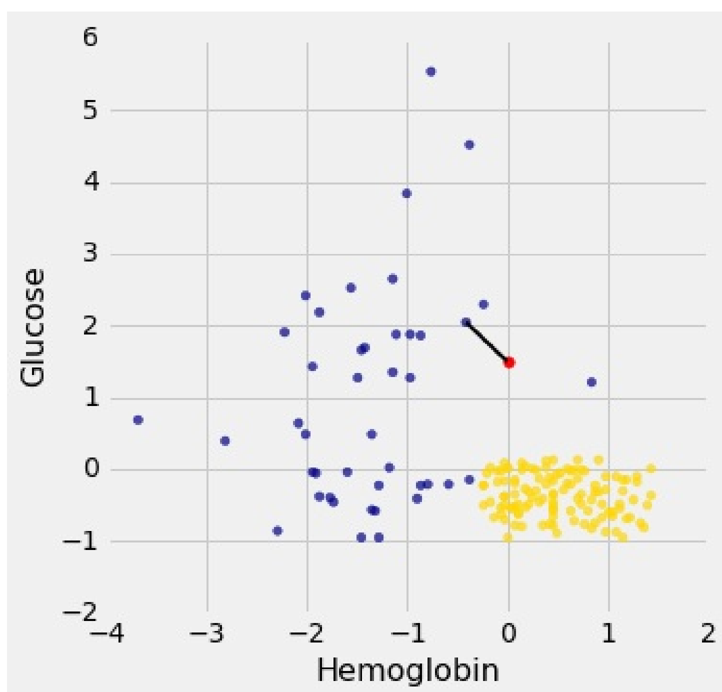
最近邻分类器

我们可能寻找很多种模式，还有很多分类算法。但是我会告诉你一个算法，它拥有令人惊讶的效果。它被称为最近邻分类。这是它的思路。如果我们有爱丽丝的血红蛋白和血糖数值，我们可以把她放在这个散点图的某个地方；血红蛋白是她的 `x` 坐标，血糖是她的 `y` 坐标。现在，为了预测她是否有 CKD，我们在散点图中找到最近的点，检查它是蓝色还是金色；我们预测爱丽丝应该接受与该患者相同的诊断。

换句话说，为了将 Alice 划分为 CKD 与否，我们在训练集中找到与 Alice “最近”的患者，然后将该患者的诊断用作对 Alice 的预测。直觉上，如果散点图中的两个点彼此靠近，那么相应的测量结果非常相似，所以我们可能会预计，他们（更可能）得到相同的诊断。我们不知道 Alice 的诊断，但是我们知道训练集中所有病人的诊断，所以我们在训练集中找到与 Alice 最相似的病人，并利用病人的诊断来预测 Alice 的诊断。

在下图中，红点代表爱丽丝。它与距离它最近的点由一条黑线相连，即训练集中最近邻。该图由一个名为 `show_closest` 的函数绘制。它需要一个数组，代表 Alice 点的 `x和` y`坐标。改变它们来查看最近的点如何改变！特别注意最近的点是蓝色，以及金色的时候。

```
# In this example, Alice's Hemoglobin attribute is 0 and her Glucose is 1.5.
alice = make_array(0, 1.5)
show_closest(alice)
```



因此，我们的最近邻分类器是这样工作的：

- 找到训练集中离新点最近的点。
- 如果最近的点是“CKD”点，则将新点划分为“CKD”。如果最近的点是“非 CKD”点，则将新点划分为“非 CKD”。

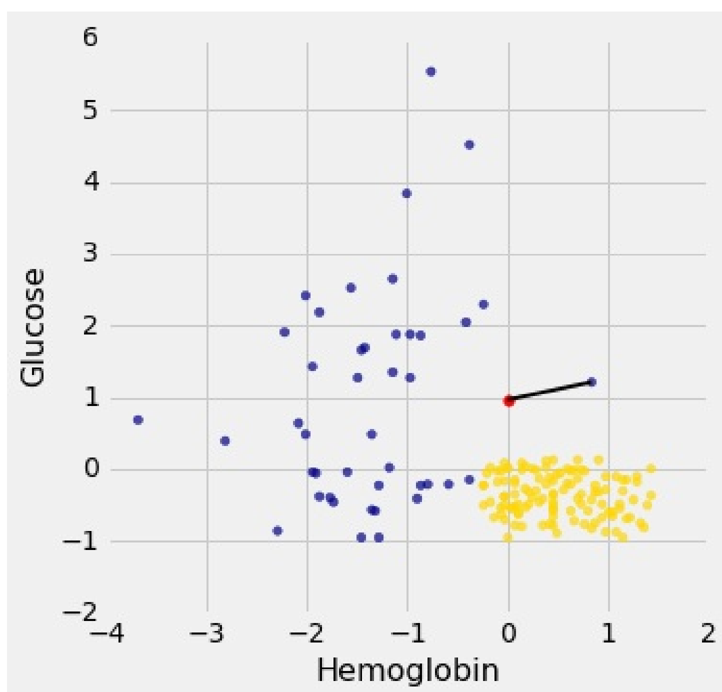
散点图表明这个最近邻分类器应该相当准确。右下角的点倾向于接受“非 CKD”的诊断，因为他们的最近邻是一个金点。其余的点倾向于接受“CKD”诊断，因为他们的最近邻是蓝点。所以这个例子中，最近邻策略似乎很好地捕捉了我们的直觉。

决策边界

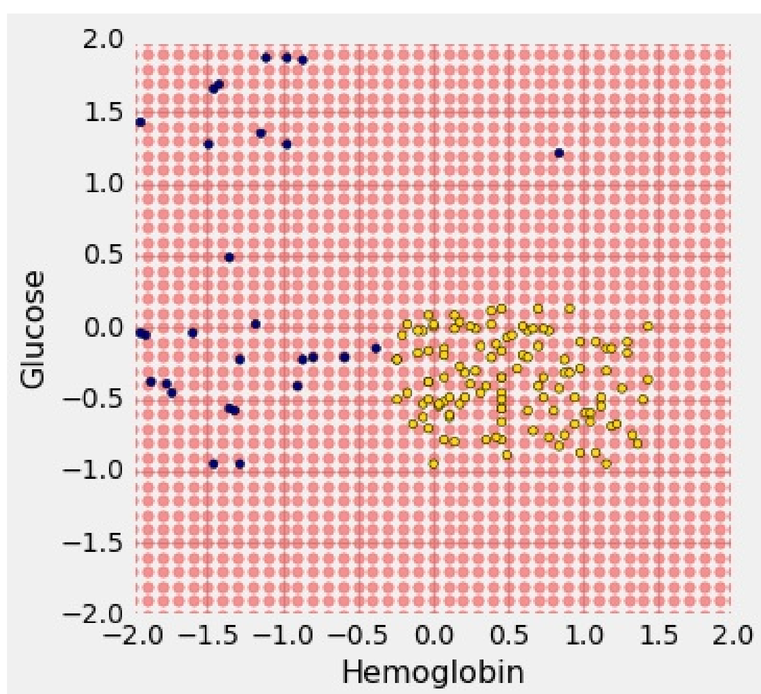
有时一种分类器可视化的实用方法是，绘制出分类器预测“CKD”的几种属性，以及预测“非 CKD”的几种。我们最终得到两者之间的边界，边界一侧的点将被划分为“CKD”，而另一侧的点将划分为“非 CKD”。这个边界称为决策边界。每个不同的分类器将有不同的决策边界；决策边界只是一种方法，用于可视化分类器实用什么标准来对点分类。

例如，假设爱丽丝的点坐标是 `(0, 1.5)` 。注意最近邻是蓝色的。现在尝试减少点的高度（ `y` 坐标）。你会看到，在 `y = 0.95` 左右，最近邻从蓝色变为金色。

```
alice = make_array(0, 0.97)
show_closest(alice)
```



这里有数百个未分类的新点，都是红色的。

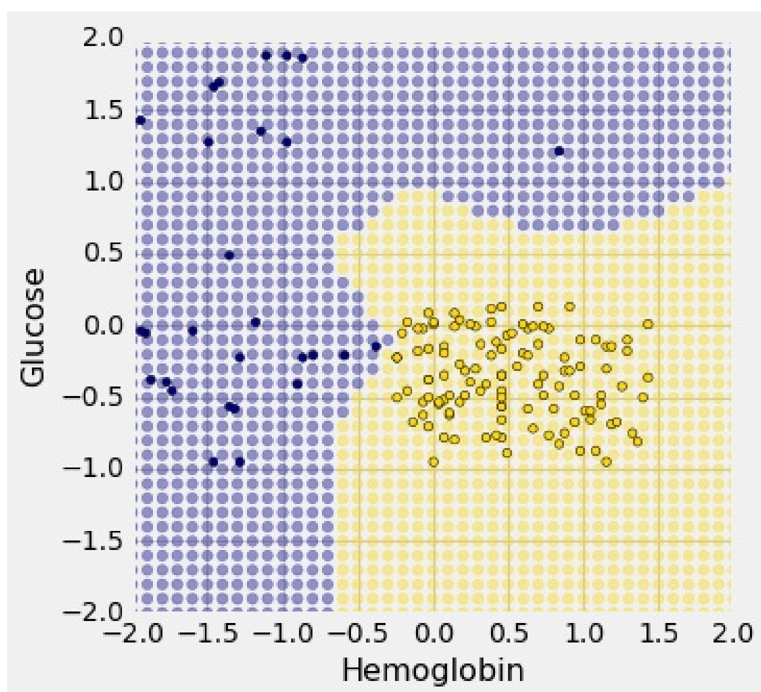


每个红点在训练集中都有一个最近邻（与之前的蓝点和金点相同）。对于一些红点，你可以很容易地判断最近邻是蓝色还是金色。对于其他点来说，通过眼睛来做出决定更为棘手。那些是靠近决策边界的点。

但是计算机可以很容易地确定每个点的最近邻。那么让我们将我们的最近邻分类器应用于每个红点：

对于每个红点，它必须找到训练集中最近的点；它必须将红点的颜色改变为最近邻的颜色。

结果图显示哪些点将划分为“CKD”（全部为蓝色），或者“非 CKD”（全部为金色）。

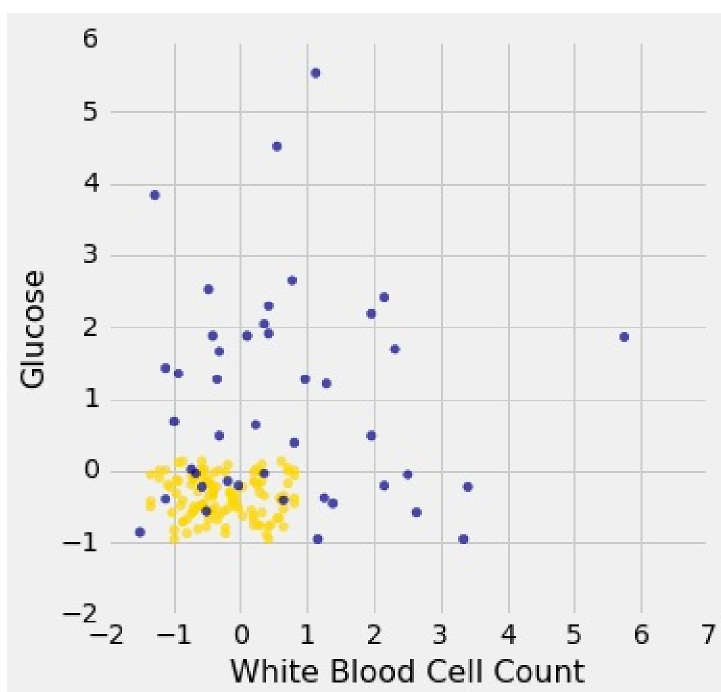


决策边界是分类器从将红点转换为蓝色变成金色的地方。

KNN

然而，两个类别的分类并不总是那么清晰。例如，假设我们不用血红蛋白水平而是看白细胞计数。看看会发生什么：

```
ckd.scatter('White Blood Cell Count', 'Glucose', colors='Color')
```



如你所见，非 CKD 个体都聚集在左下角。大多数 CKD 患者在该簇的上方或右侧，但不是全部。上图左下角有一些 CKD 患者（分散在金簇中的少数蓝点表示）。这意味着你不能从这两个检测结果确定，某些人是否拥有 CKD。

如果提供爱丽丝的血糖水平和白细胞计数，我们可以预测她是否患有慢性肾病嘛？是的，我们可以做一个预测，但是我们不应该期望它是 100% 准确的。直觉上，似乎存在预测的自然策略：绘制 Alice 在散点图中的位置；如果她在左下角，则预测她没有 CKD，否则预测她有 CKD。

这并不完美 - 我们的预测有时是错误的。（请花点时间思考一下，会把哪些患者弄错？）上面的散点图表明，CKD 患者的葡萄糖和白细胞水平有时与没有 CKD 的患者相同，因此任何分类器都是不可避免地会对他们做出错误的预测。

我们可以在计算机上自动化吗？那么，最近邻分类器也是一个合理的选择。花点时间思考一下：它的预测与上述直觉策略的预测相比如何？他们什么时候会不同？

它的预测与我们的直觉策略非常相似，但偶尔会做出不同的预测。特别是，如果爱丽丝的血液检测结果恰好把她放在左下角的一个蓝点附近，那么这个直观的策略就可能预测“非 CKD”，而最近邻的分类器会预测“CKD”。

最近邻分类器有一个简单的推广，修正了这个异常。它被称为 K 最近邻分类器。为了预测爱丽丝的诊断，我们不仅仅查看靠近她的一个邻居，而是查看靠近她的三个点，并用这三个点中的每一个点的诊断来预测艾丽丝的诊断。特别是，我们将使用这 3 个诊断中的大部分值作为我们对 Alice 诊断的预测。当然，数字 3 没有什么特别之处：我们可以使用 4 或 5 或更多。（选择一个奇数通常是很方便的，所以我们不需要处理相等）。一般来说，我们选择一个数字 `k`，而我们对 Alice 的预测诊断是基于训练集中最接近爱丽丝的 `k` 个点。直观来说，这些是血液测试结果与爱丽丝最相似的 `k` 个患者，因此使用他们的诊断来预测爱丽丝的诊断似乎是合理的。

训练和测试

我们最近的邻居分类器有多好？要回答这个问题，我们需要知道我们的分类有多正确。如果患者患有慢性肾脏疾病，那么我们的分类器有多可能将其选出来呢？

如果病人在我们的训练集中，我们可以立即找到。我们已经知道病人位于什么类别，所以我们可以比较我们的预测和病人的真实类别。

但是分类器的重点在于对未在训练集中的新患者进行预测。我们不知道这些病人位于什么类别，但我们可以根据分类器做出预测。如何知道预测是否正确？

一种方法是等待患者之后的医学检查，然后检查我们的预测是否与检查结果一致。用这种方法，当我们可以说我们的预测有多准确的时候，它就不再能用于帮助病人了。

相反，我们将在一些真实类别已知的病人上尝试我们的分类器。然后，我们将计算分类器正确的时间比例。这个比例将作为我们分类器准确预测的所有新患者的比例的估计值。这就是所谓的测试。

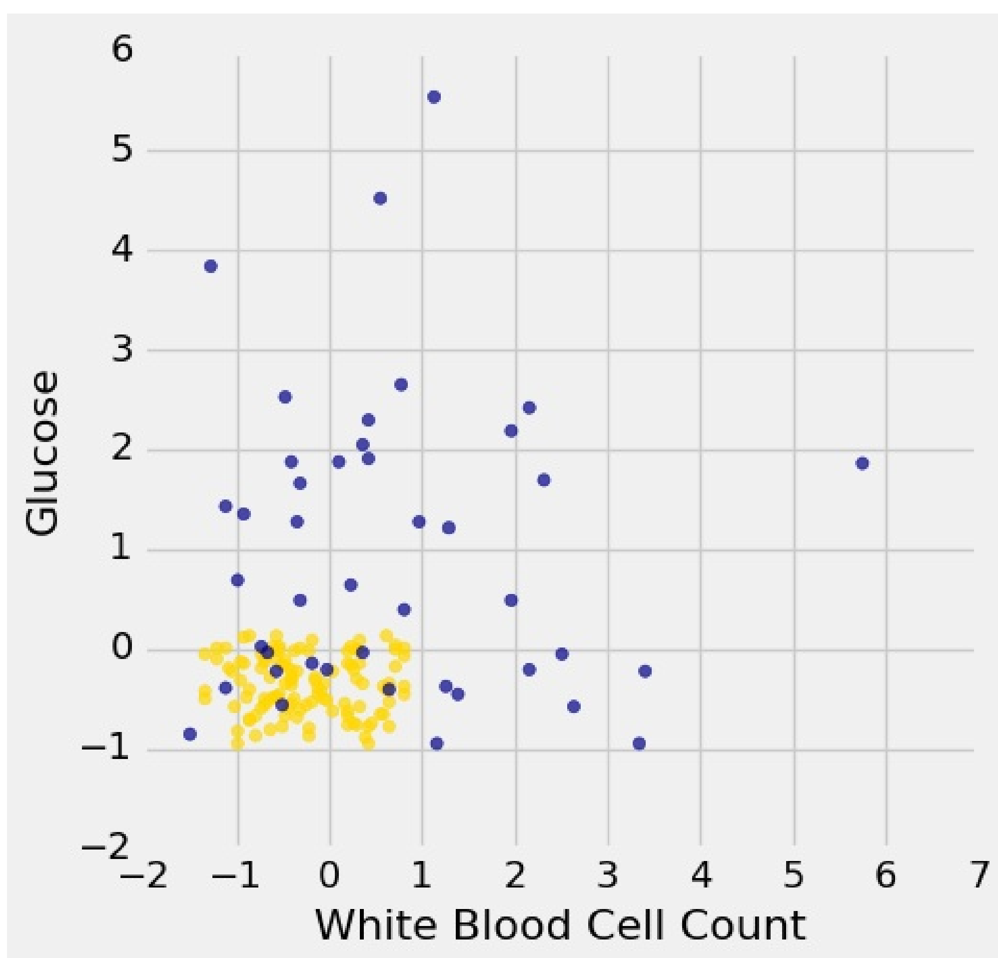
过于乐观的“测试”

训练集提供了一组非常吸引人的患者，我们在它们上测试我们的分类器，因为我们可以知道训练集中每个患者的分类。

但是，我们要小心，如果我们走这条道路，前面就会有隐患。一个例子会告诉我们为什么。

假设我们使用 1 邻近分类器，根据血糖和白细胞计数来预测患者是否患有慢性肾病。

```
ckd.scatter('White Blood Cell Count', 'Glucose', colors='Color')
```



之前，我们说我们预计得到一些分类错误，因为在左下方有一些蓝色和金色的点。

但是训练集中的点，也就是已经在散点图上的点呢？我们会把它们误分类吗？

答案是否。请记住，1 最近邻分类寻找训练集中离被分类点最近的点。那么，如果被分类的点已经在训练集中，那么它在训练集中的最近邻就是它自己！因此它将被划分为自己的颜色，这将是正确的，因为训练集中的每个点都已经被正确着色。

换句话说，如果我们使用我们的训练集来“测试”我们的 1 邻近分类器，分类器将以 100% 的几率内通过测试。

任务完成。多好的分类器！

不，不是。正如我们前面提到的，左下角的一个新点很容易被误分类。“100% 准确”是一个很好的梦想，而它持续。

这个例子的教训是不要使用训练集来测试基于它的分类器。

生成测试集

在前面的章节中，我们看到可以使用随机抽样来估计符合一定标准的总体中的个体比例。不幸的是，我们刚刚看到训练集不像所有患者总体中的随机样本，在一个重要的方面：我们的分类器正确猜测训练集中的个体，比例高于总体中的个体。

当我们计算数值参数的置信区间时，我们希望从一个总体中得到许多新的随机样本，但是我们只能访问一个样本。我们通过从我们的样本中自举重采样来解决这个问题。

我们将使用一个类似的想法来测试我们的分类器。我们将从原始训练集中创建两个样本，将其中一个样本作为我们的训练集，另一个用于测试。

所以我们将有三组个体：

- 训练集，我们可以对它进行任何大量的探索来建立我们的分类器
- 一个单独的测试集，在它上面测试我们的分类器，看看分类的正确比例是多少
- 个体的底层总体，我们不了解它；我们的希望是我们的分类器对于这些个体也会成功，就像我们的测试集一样。

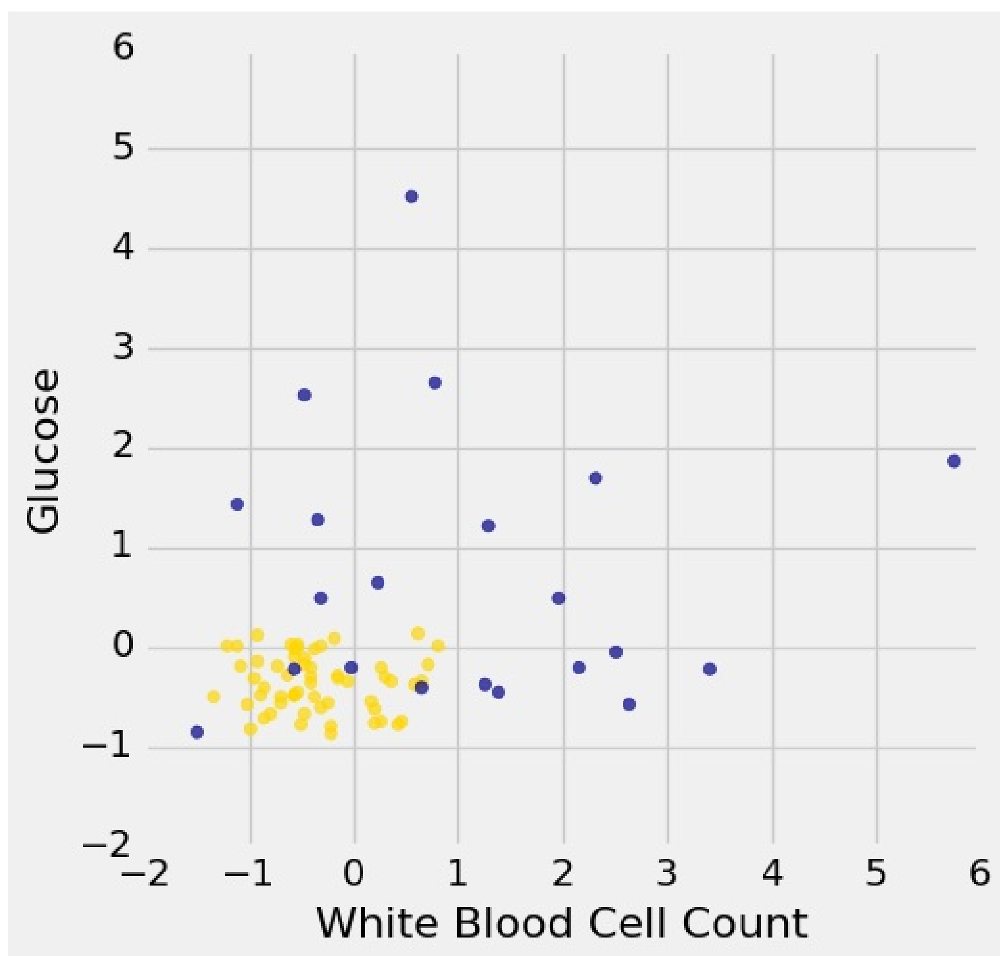
如何生成训练和测试集？你猜对了 - 我们会随机选择。

`ckd` 有 158 个个体。让我们将它们随机的一半用于训练，另一半用于测试。为此，我们将打乱所有行，把前 79 个作为训练集，其余的 79 个用于测试。

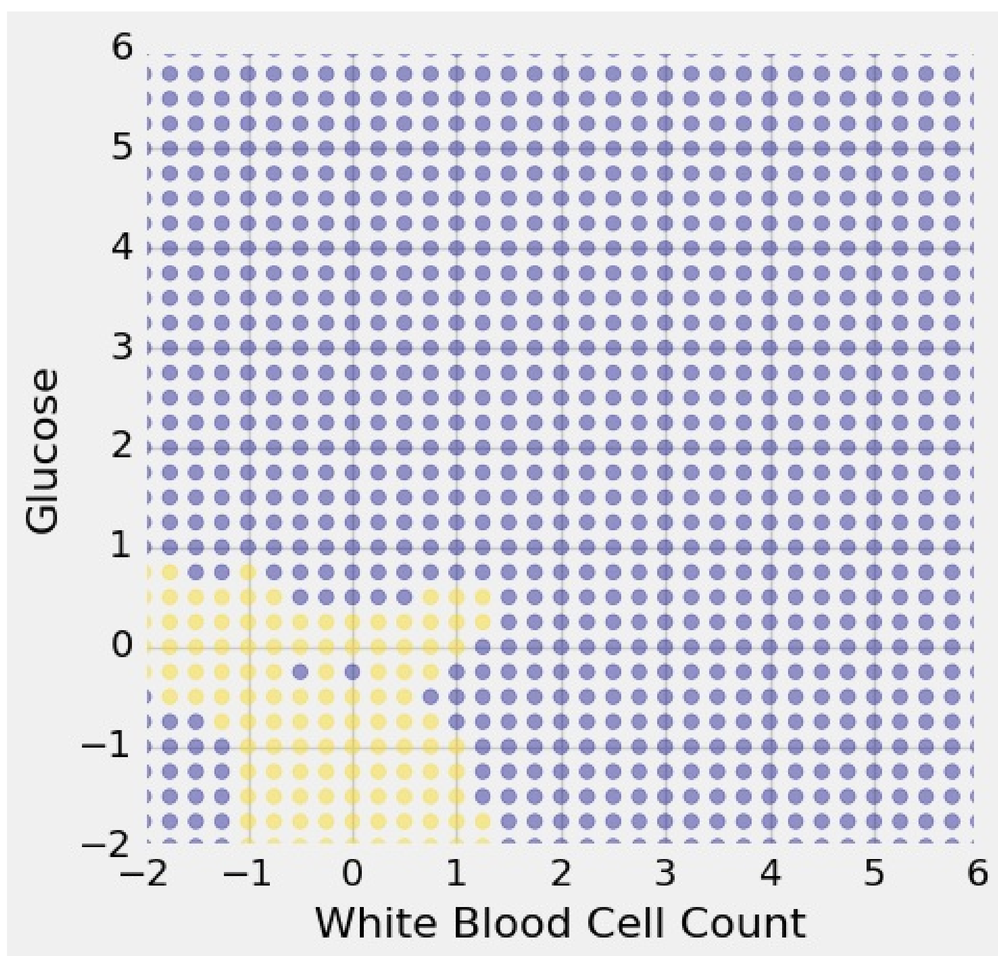
```
shuffled_ckd = ckd.sample(with_replacement=False)
training = shuffled_ckd.take(np.arange(79))
testing = shuffled_ckd.take(np.arange(79, 158))
```

现在让我们基于训练样本中的点构造我们的分类器：

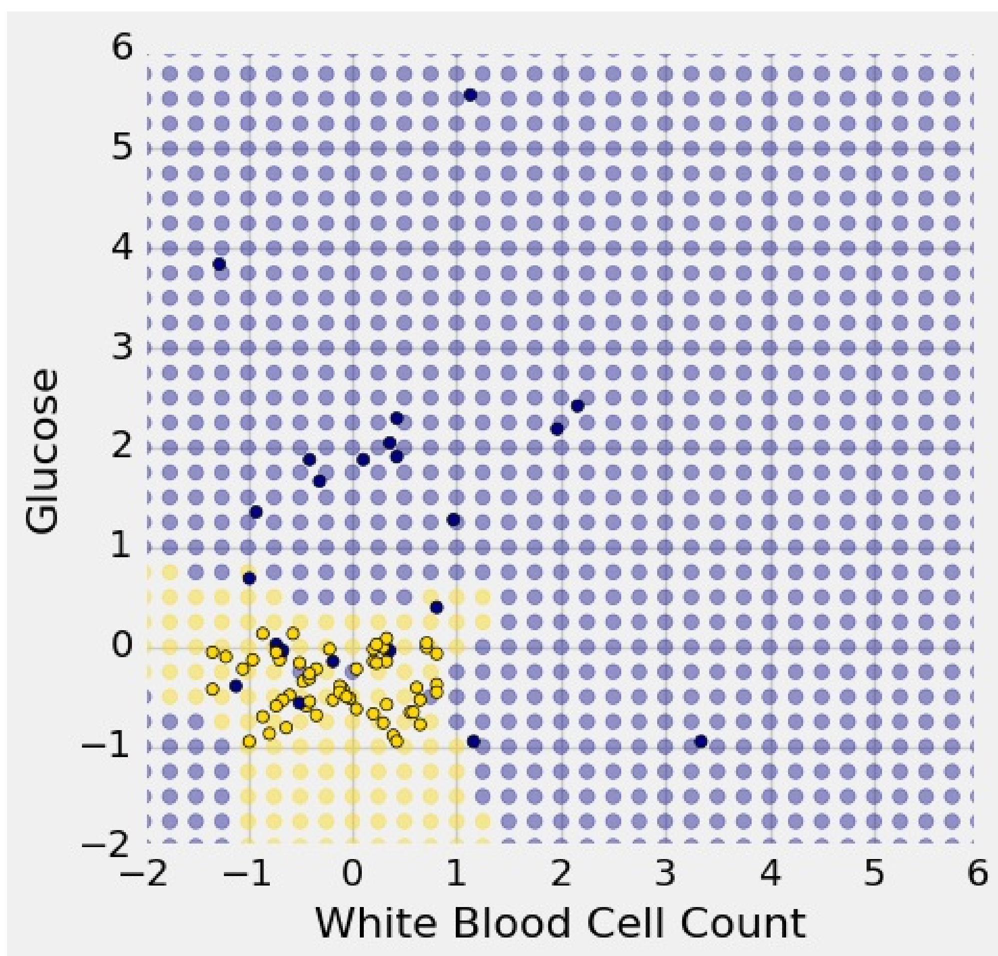
```
training.scatter('White Blood Cell Count', 'Glucose', colors='Color')
plt.xlim(-2, 6)
plt.ylim(-2, 6);
```



我们得到以下分类区域和决策边界：



把测试数据放在这个图上，你可以立刻看到分类器对于几乎所有的点都正确，但也有一些错误。 例如，测试集的一些蓝点落在分类器的金色区域。



尽管存在一些错误，但分类器看起来在测试集上表现得相当好。 假设原始样本是从底层总体中随机抽取的，我们希望分类器在整个总体上具有相似的准确性，因为测试集是从原始样本中随机选取的。

表的行

现在我们对最近邻分类有一个定性的了解，是时候实现我们的分类器了。

在本章之前，我们主要处理表格的单列。 但现在我们必须看看一个个体是否“接近”另一个个体。 个体数据包含在表格的行中。

那么让我们首先仔细看一下行。

这里是原始表格 `ckd` ，包含慢性肾病患者资料。

```
ckd = Table.read_table('ckd.csv').relabeled('Blood Glucose Random', 'Glucose')
```

对应第一个患者的数据在表中第 0 行，与 Python 的索引系统一致。 `Table` 的 `row` 方法将行索引作为其参数来访问行。

```
ckd.row(0)
Row(Age=48, Blood Pressure=70, Specific Gravity=1.0049999999999999, Albumin=4, Sugar=0
, Red Blood Cells='normal', Pus Cell='abnormal', Pus Cell clumps='present', Bacteria='
notpresent', Glucose=117, Blood Urea=56, Serum Creatinine=3.7999999999999998, Sodium=1
11, Potassium=2.5, Hemoglobin=11.199999999999999, Packed Cell Volume=32, White Blood C
ell Count=6700, Red Blood Cell Count=3.8999999999999999, Hypertension='yes', Diabetes
Mellitus='no', Coronary Artery Disease='no', Appetite='poor', Pedal Edema='yes', Anemi
a='yes', Class=1)
```

行拥有自己的数据类型：它们是行对象。注意屏幕不仅显示行中的值，还显示相应列的标签。

行通常不是数组，因为它们的元素可以是不同的类型。例如，上面那行的一些元素是字符串（如 `'abnormal'` ），有些是数字。所以行不能被转换成数组。

但是，行与数组有一些特征。你可以使用 `item` 来访问行中的特定元素。例如，要访问患者 0 的白蛋白水平，我们可以查看上面那行的打印输出中的标签，发现它是第 3 项：

```
ckd.row(0).item(3)
4
```

将行转换为数组（可能的时候）

元素都是数字（或都是字符串）的行可以转换为数组。将行转换为数组可以让我们访问算术运算和其他漂亮的 NumPy 函数，所以它通常很有用。

回想一下，在上一节中，我们试图根据血红蛋白和血糖两个属性将患者划分为“CKD”或“非 CKD”，这两个属性都是以标准单位测量的。

```
ckd = Table().with_columns(
    'Hemoglobin', standard_units(ckd.column('Hemoglobin')),
    'Glucose', standard_units(ckd.column('Glucose')),
    'Class', ckd.column('Class')
)

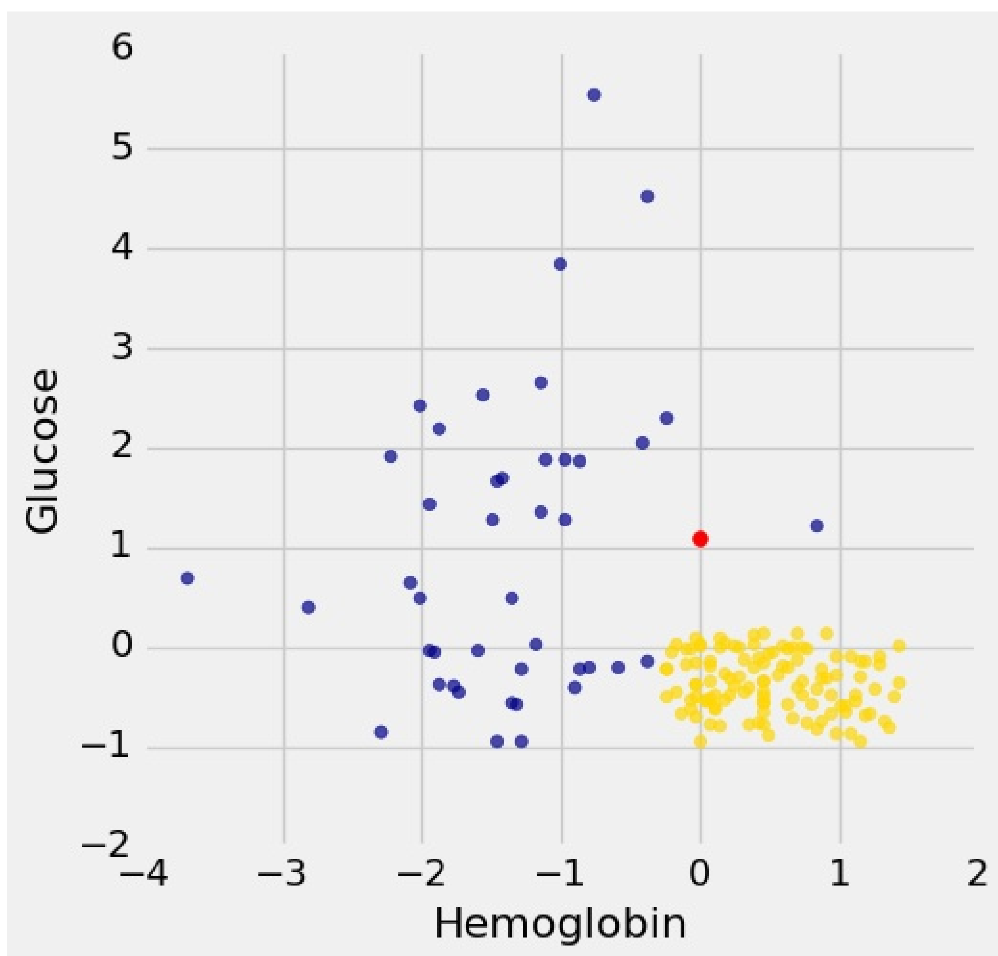
color_table = Table().with_columns(
    'Class', make_array(1, 0),
    'Color', make_array('darkblue', 'gold')
)
ckd = ckd.join('Class', color_table)
ckd
```

Class	Hemoglobin	Glucose	Color
0	0.456884	0.133751	gold
0	1.153	-0.947597	gold
0	0.770138	-0.762223	gold
0	0.596108	-0.190654	gold
0	-0.239236	-0.49961	gold
0	-0.0304002	-0.159758	gold
0	0.282854	-0.00527964	gold
0	0.108824	-0.623193	gold
0	0.0740178	-0.515058	gold
0	0.83975	-0.422371	gold

（省略了 148 行）

下面是两个属性的散点图，以及新患者 Alice 对应的红点。 她的血红蛋白值是 0（即平均值）和血糖为 1.1（即比平均值高 1.1 个 SD）。

```
alice = make_array(0, 1.1)
ckd.scatter('Hemoglobin', 'Glucose', colors='Color')
plots.scatter(alice.item(0), alice.item(1), color='red', s=30);
```



为了找到 Alice 点和其他点之间的距离，我们只需要属性的值：

```
ckd_attributes = ckd.select('Hemoglobin', 'Glucose')
ckd_attributes
```

Hemoglobin	Glucose
0.456884	0.133751
1.153	-0.947597
0.770138	-0.762223
0.596108	-0.190654
-0.239236	-0.49961
-0.0304002	-0.159758
0.282854	-0.00527964
0.108824	-0.623193
0.0740178	-0.515058
0.83975	-0.422371

（省略了 148 行）

每行由我们的训练样本中的一个点的坐标组成。 由于行现在只包含数值，因此可以将它们转换为数组。 为此，我们使用函数 `np.array`，将任何类型的有序对象（如行）转换为数组。（我们的老朋友 `make_array` 用于创建数组，而不是用于将其他类型的序列转换为数组。）

```
ckd_attributes.row(3)
Row(Hemoglobin=0.5961076648232668 3, Glucose=-0.19065363034327712)
np.array(ckd_attributes.row(3))
array([ 0.59610766, -0.19065363])
```

这非常方便，因为我们现在可以在每行的数据上使用数组操作了。

只有两个属性时点的距离

我们需要做的主要计算是，找出 Alice 的点与其他点之间的距离。 为此，我们需要的第一件事就是计算任意一对点之间的距离。

我们如何实现呢？ 在二维空间中，这非常简单。 如果我们在坐标 `(x0, y0)` 处有一个点，而在 `(x1, y1)` 处有另一个点，则它们之间的距离是：

$$D = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

（这是从哪里来的？它来自勾股定理：我们有一个直角三角形，边长为 `x0 - x1` 和 `y0 - y1`，我们想要求出斜边的长度。）

在下一节中，我们将看到，当存在两个以上的属性时，这个公式有个直接的扩展。 现在，让我们使用公式和数组操作来求出 Alice 和第 3 行病人的距离。

```
patient3 = np.array(ckd_attributes.row(3))
alice, patient3
(array([ 0. ,  1.1]), array([ 0.59610766, -0.19065363]))
distance = np.sqrt(np.sum((alice - patient3)**2))
distance
1.4216649188818471
```

我们需要 Alice 和一堆点之间的距离，所以让我们写一个称为距离的函数来计算任意一对点之间的距离。 该函数将接受两个数组，每个数组包含一个点的 `(x, y)` 坐标。 （记住，那些实际上是患者的血红蛋白和血糖水平。）

```
def distance(point1, point2):
    """Returns the Euclidean distance between point1 and point2.

    Each argument is an array containing the coordinates of a point."""
    return np.sqrt(np.sum((point1 - point2)**2))
distance(alice, patient3)
1.4216649188818471
```

我们已经开始建立我们的分类器：距离函数是第一个积木。 现在让我们来研究下一个片段。

在整个行上使用 `apply`

回想一下，如果要将函数应用于表的列的每个元素，一种方法是调用 `table_name.apply(function_name, column_label)` 。 当我们在列的每个元素上调用该函数时，它求值为由函数返回值组成的数组。所以数组的每个条目都基于表的相应行。

如果使用 `apply` 而不指定列标签，则整行将传递给该函数。 让我们在一个非常小的表格上，看看它的工作原理，表格包含训练样本中前五个患者的信息。

```
t = ckd_attributes.take(np.arange(5))
t
```

Hemoglobin	Glucose
0.456884	0.133751
1.153	-0.947597
0.770138	-0.762223
0.596108	-0.190654
-0.239236	-0.49961

举个例子，假设对于每个病人，我们都想知道他们最不寻常的属性是多么的不寻常。 具体而言，如果患者的血红蛋白水平远高于其血糖水平，我们想知道它离平均值有多远。 如果她的血糖水平远远高于她的血红蛋白水平，那么我们想知道它离平均值有多远。

这与获取两个量的绝对值的最大值是一样的。 为了为特定的行执行此操作，我们可以将行转换为数组并使用数组操作。

```
def max_abs(row):
    return np.max(np.abs(np.array(row)))
max_abs(t.row(4))
0.49961028259186968
```

现在我们可以将 `max_abs` 应用于 `t` 表的每一行：

```
t.apply(max_abs)
array([ 0.4568837 ,  1.15300352,  0.77013762,  0.59610766,  0.49961028])
```

这种使用 `apply` 的方式帮助我们创建分类器的下一个积木。

Alice 的 K 最近邻

如果我们想使用 K 最近邻分类器来划分 Alice，我们必须确定她的 K 个最近邻。 这个过程中的步骤是什么？ 假设 `k = 5` 。 然后这些步骤是：

- 步骤 1：的是 Alice 与训练样本中每个点之间的距离。
- 步骤 2：按照距离的升序对数据表进行排序。
- 步骤 3：取得有序表的前 5 行。

步骤 2 和步骤 3 似乎很简单，只要我们有了距离。 那么我们来关注步骤 1。

这是爱丽丝：

```
alice
array([ 0. ,  1.1])
```

我们需要一个函数，它可以求出 Alice 和另一个点之间的距离，它的坐标包含在一行中。 `distance` 函数返回任意两点之间的距离，他们的坐标位于数组中。 我们可以使用它来定义 `distance_from_alice` ，它将一行作为参数，并返回该行与 Alice 之间的距离。

```
def distance_from_alice(row):
    """Returns distance between Alice and a row of the attributes table"""
    return distance(alice, np.array(row))
distance_from_alice(ckd_attributes.row(3))
1.4216649188818471
```

现在我们可以调用 `apply` ，将 `distance_from_alice` 函数应用于 `ckd_attributes` 的每一行，第一步完成了。

```
distances = ckd_attributes.apply(distance_from_alice)
ckd_with_distances = ckd.with_column('Distance from Alice', distances)
ckd_with_distances
```

Class	Hemoglobin	Glucose	Color	Distance from Alice
0	0.456884	0.133751	gold	1.06882
0	1.153	-0.947597	gold	2.34991
0	0.770138	-0.762223	gold	2.01519
0	0.596108	-0.190654	gold	1.42166
0	-0.239236	-0.49961	gold	1.6174
0	-0.0304002	-0.159758	gold	1.26012
0	0.282854	-0.00527964	gold	1.1409
0	0.108824	-0.623193	gold	1.72663
0	0.0740178	-0.515058	gold	1.61675
0	0.83975	-0.422371	gold	1.73862

（省略了 148 行）

对于步骤 2，让我们以距离的升序对表排序：

```
sorted_by_distance = ckd_with_distances.sort('Distance from Alice')
sorted_by_distance
```

Class	Hemoglobin	Glucose	Color	Distance from Alice
1	0.83975	1.2151	darkblue	0.847601
1	-0.970162	1.27689	darkblue	0.986156
0	-0.0304002	0.0874074	gold	1.01305
0	0.14363	0.0874074	gold	1.02273
1	-0.413266	2.04928	darkblue	1.03534
0	0.387272	0.118303	gold	1.05532
0	0.456884	0.133751	gold	1.06882
0	0.178436	0.0410639	gold	1.07386
0	0.00440582	0.025616	gold	1.07439
0	-0.169624	0.025616	gold	1.08769

（省略了 148 行）

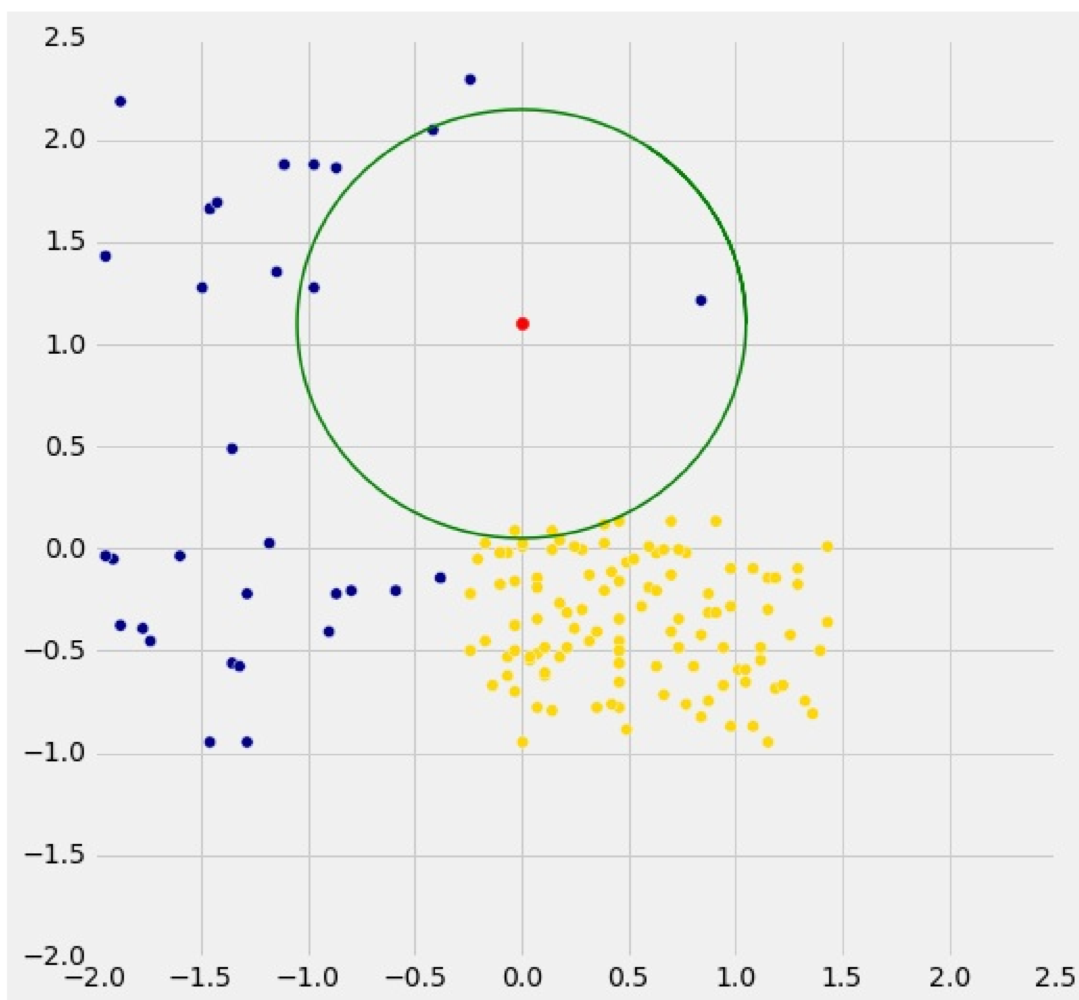
步骤 3：前五行对应 Alice 的五个最近邻；你可以将五替换为任意正整数。

```
alice_5_nearest_neighbors = sorted_by_distance.take(np.arange(5))
alice_5_nearest_neighbors
```

Class	Hemoglobin	Glucose	Color	Distance from Alice
1	0.83975	1.2151	darkblue	0.847601
1	-0.970162	1.27689	darkblue	0.986156
0	-0.0304002	0.0874074	gold	1.01305
0	0.14363	0.0874074	gold	1.02273
1	-0.413266	2.04928	darkblue	1.03534

爱丽丝五个最近邻中有三个是蓝点，两个是金点。所以 5 邻近的分类器会把爱丽丝划分为蓝色：它可能预测爱丽丝有慢性肾病。

下面的图片放大了爱丽丝和她五个最近邻。这两个金点就在红点正下方的圆圈内。分类器说，爱丽丝更像她身边的三个蓝点。



我们正在实现我们的 K 最近邻分类器。 在接下来的两节中，我们将把它放在一起并评估其准确性。

实现分类器

现在我们准备基于多个属性实现 K 最近邻分类器。 到目前为止，我们只使用了两个属性，以便可视化。 但通常预测将基于许多属性。 这里是一个例子，显示了多个属性可能比两个更好。

钞票检测

这次我们来看看，预测钞票（例如 20 美元钞票）是伪造还是合法的。 研究人员根据许多单个钞票的照片，为我们汇集了一套数据集：一些是假冒的，一些是合法的。 他们从每张图片中计算出一些数字，使用这门课中我们无需担心的技术。 所以，对于每一张钞票，我们知道了一些数字，它们从钞票的照片以及它的类别（是否是伪造的）中计算。 让我们把它加载到一个表中，并看一下。

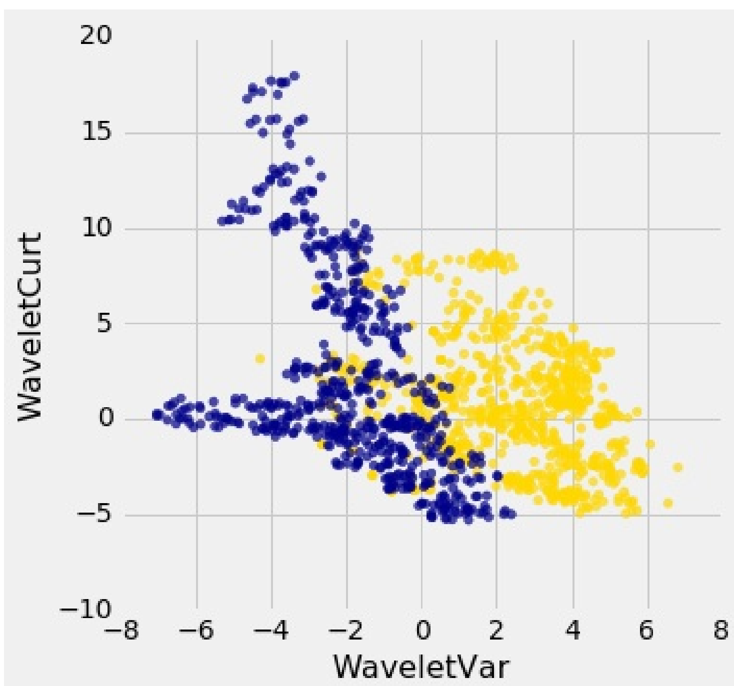
```
banknotes = Table.read_table('banknote.csv')
banknotes
```

WaveletVar	WaveletSkew	WaveletCurt	Entropy	Class
3.6216	8.6661	-2.8073	-0.44699	0
4.5459	8.1674	-2.4586	-1.4621	0
3.866	-2.6383	1.9242	0.10645	0
3.4566	9.5228	-4.0112	-3.5944	0
0.32924	-4.4552	4.5718	-0.9888	0
4.3684	9.6718	-3.9606	-3.1625	0
3.5912	3.0129	0.72888	0.56421	0
2.0922	-6.81	8.4636	-0.60216	0
3.2032	5.7588	-0.75345	-0.61251	0
1.5356	9.1772	-2.2718	-0.73535	0

（省略了 1362 行）

让我们看看，前两个数值是否告诉了我们，任何钞票是否伪造的事情。这里是散点图：

```
color_table = Table().with_columns(
    'Class', make_array(1, 0),
    'Color', make_array('darkblue', 'gold')
)
banknotes = banknotes.join('Class', color_table)
banknotes.scatter('WaveletVar', 'WaveletCurt', colors='Color')
```

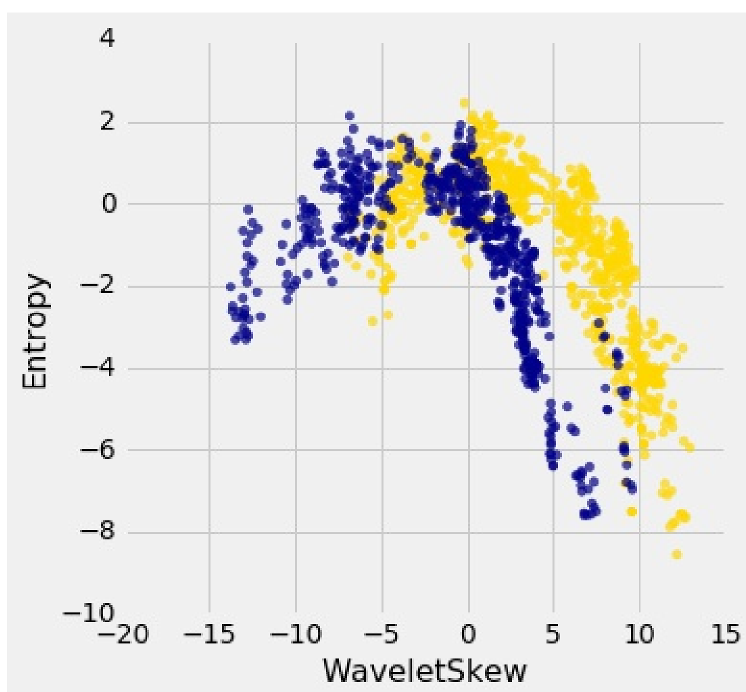


非常有趣！ 这两个测量值看起来对于预测钞票是否伪造有帮助。 然而，在这个例子中，你现在可以看到蓝色的簇和金色的簇之间有一些重叠。 这表示基于这两个数字，很难判断钞票是否合法。 不过，你可以使用 K 最近邻分类器来预测钞票的合法性。

花点时间想一想：假设我们使用 `k = 11`（是假如）。 图中的哪些部分会得到正确的结果，哪些部分会产生错误？ 决定边界是什么样子？

数据中显示的规律可能非常乱。 例如，如果使用与图像不同的一对测量值，我们可以得到以下结果：

```
banknotes.scatter('WaveletSkew', 'Entropy', colors='Color')
```



似乎存在规律，但它是非常复杂。 尽管如此， K 最近邻分类器仍然可以使用，并将有效地“发现”规律。 这说明了机器学习有多强大：它可以有效地利用规律，我们不曾预料到它，或者我们打算将其编入计算机。

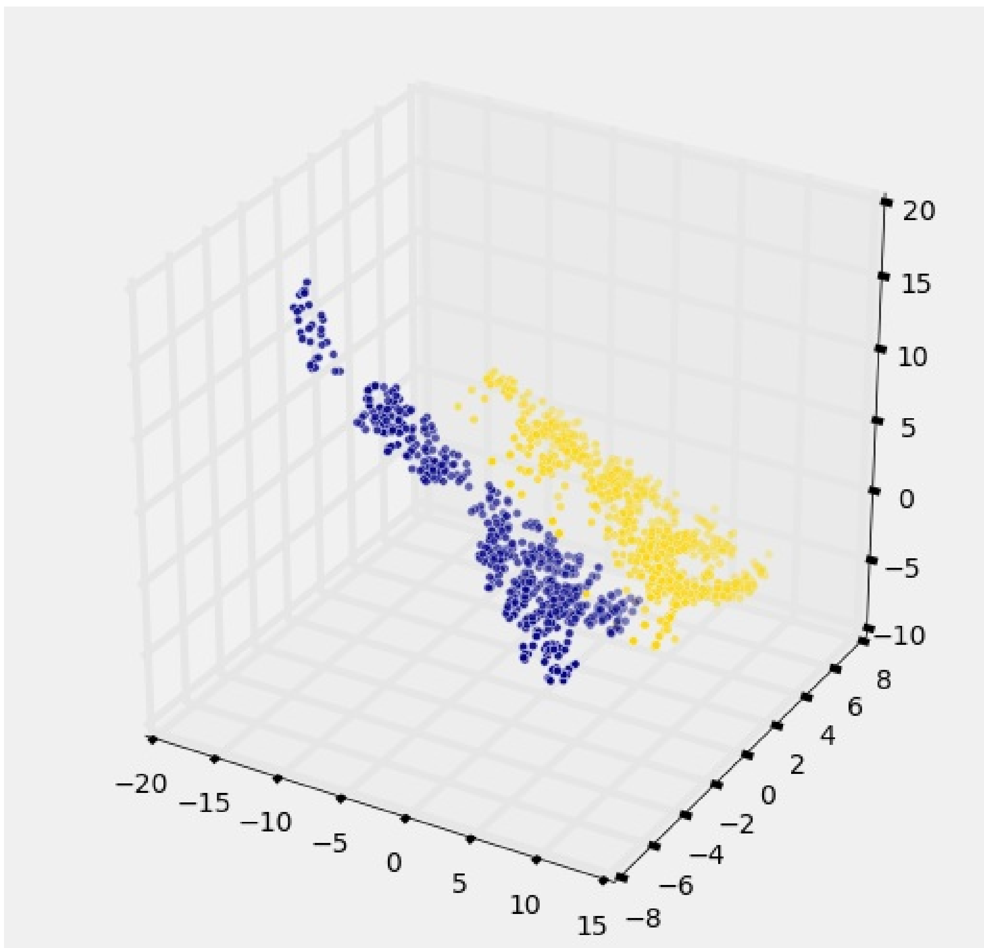
多个属性

到目前为止，我一直假设我们有两个属性，可以用来帮助我们做出预测。如果我们有两个以上呢？例如，如果我们有 3 个属性呢？

这里有一个很酷的部分：你也可以对这个案例使用同样的想法。你需要做的所有事情，就是绘制一个三维散点图，而不是二维的。你仍然可以使用 K 最近邻分类器，但现在计算 3 维而不是 2 维距离，它还是有用。可以，很酷！

事实上，2 或 3 没有什么特别之处。如果你有 4 个属性，你可以使用 4 维的 K 最近邻分类器。 5 个属性？在五维空间里工作。没有必要在这里停下来！这一切都适用于任意多的属性。你只需在非常高维的空间中工作。它变得有点奇怪 - 不可能可视化，但没关系。计算机算法推广得很好：你需要的所有事情，就是计算距离的能力，这并不难。真是亦可赛艇！

```
ax = plt.figure(figsize=(8,8)).add_subplot(111, projection='3d')
ax.scatter(banknotes.column('WaveletSkew'),
           banknotes.column('WaveletVar'),
           banknotes.column('WaveletCurt'),
           c=banknotes.column('Color'));
```



真棒！只用 2 个属性，两个簇之间有一些重叠（这意味着对于重叠中的一些点，分类器必然犯一些错误）。但是当我们使用这三个属性时，两个簇几乎没有重叠。换句话说，使用这 3 个属性的分类器比仅使用 2 个属性的分类器更精确。

这是分类中的普遍现象。每个属性都可能会给你提供新的信息，所以更多的属性有时可以帮助你建立一个更好的分类器。当然开销是，现在我们必须收集更多的信息来衡量每个属性的值，但是如果这个开销显着提高了我们的分类器的精度，那么它可能非常值得。

综上所述：你现在知道如何使用 K 最近邻分类，预测是与否的问题的答案，基于一些属性值，假设你有一个带有样本的训练集，其中正确的预测已知。总的路线图是这样的：

找出一些属性，你认为可能帮助你预测问题的答案。 收集一组训练样本，其中你知道属性值以及正确预测。 为了预测未来，测量属性的值，然后使用 K 最近邻分类来预测问题的答案。

多维距离

我们知道如何在二维空间中计算距离。 如果我们在坐标 `(x0, y0)` 处有一个点，而在 `(x1, y1)` 处有另一个点，则它们之间的距离是：

$$D = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

在三维空间中，点是 `(x0, y0, z0)` 和 `(x1, y1, z1)` ，它们之间的距离公式为：

$$D = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2 + (z_0 - z_1)^2}$$

在 N 维空间中，东西有点难以可视化，但我想你可以看到公式是如何推广的：我们总结每个独立坐标差的平方，然后取平方根。

在最后一节中，我们定义了函数 `distance` 返回两点之间距离。 我们在二维中使用它，但好消息是函数并不关心有多少维！ 它只是将两个坐标数组相减（无论数组有多长），求差值的平方并加起来，然后取平方根。 我们不必更改代码就可以在多个维度上工作。

```
def distance(point1, point2):
    """Returns the distance between point1 and point2
    where each argument is an array
    consisting of the coordinates of the point"""
    return np.sqrt(np.sum((point1 - point2)**2))
```

我们在这个新的数据集上使用它。 `wine` 表含有 178 种不同的意大利葡萄酒的化学成分。 这些类别是葡萄品种，称为品种。 有三个类别，但我们只看看是否可以把第一类和其他两个类别分开。

```
wine = Table.read_table('wine.csv')

# For converting Class to binary

def is_one(x):
    if x == 1:
        return 1
    else:
        return 0

wine = wine.with_column('Class', wine.apply(is_one, 0))
wine
```

Class	Alcohol	Malic Acid	Ash	Alcalinity of Ash	Magnesium	Total Phenols	Flavanc
1	14.23	1.71	2.43	15.6	127	2.8	3.06
1	13.2	1.78	2.14	11.2	100	2.65	2.76
1	13.16	2.36	2.67	18.6	101	2.8	3.24
1	14.37	1.95	2.5	16.8	113	3.85	3.49
1	13.24	2.59	2.87	21	118	2.8	2.69
1	14.2	1.76	2.45	15.2	112	3.27	3.39
1	14.39	1.87	2.45	14.6	96	2.5	2.52
1	14.06	2.15	2.61	17.6	121	2.6	2.51
1	14.83	1.64	2.17	14	97	2.8	2.98
1	13.86	1.35	2.27	16	98	2.98	3.15

前两种葡萄酒都属于第一类。为了找到它们之间的距离，我们首先需要一个只有属性的表格：

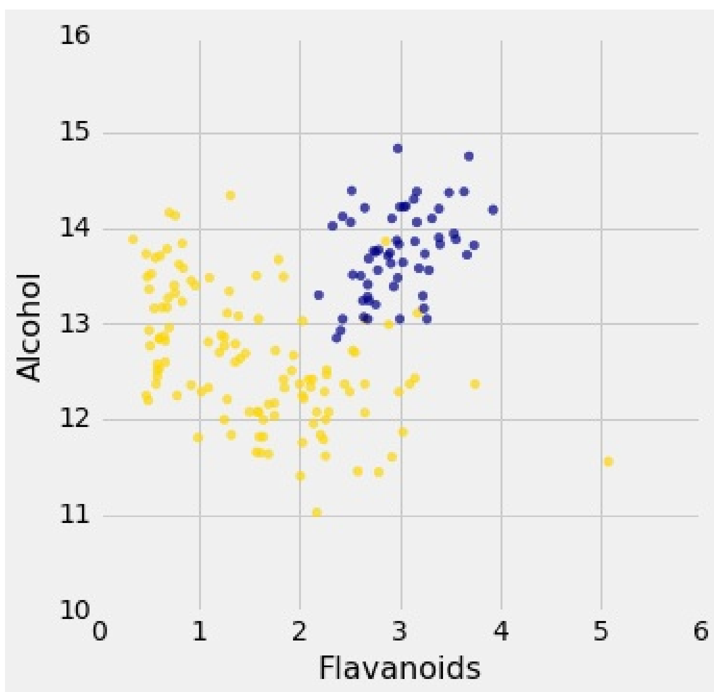
```
wine_attributes = wine.drop('Class')
distance(np.array(wine_attributes.row(0)), np.array(wine_attributes.row(1)))
31.265012394048398
```

中的最后一个葡萄酒是第零类。它与第一个葡萄酒的距离是：

```
distance(np.array(wine_attributes.row(0)), np.array(wine_attributes.row(177)))
506.05936766351834
```

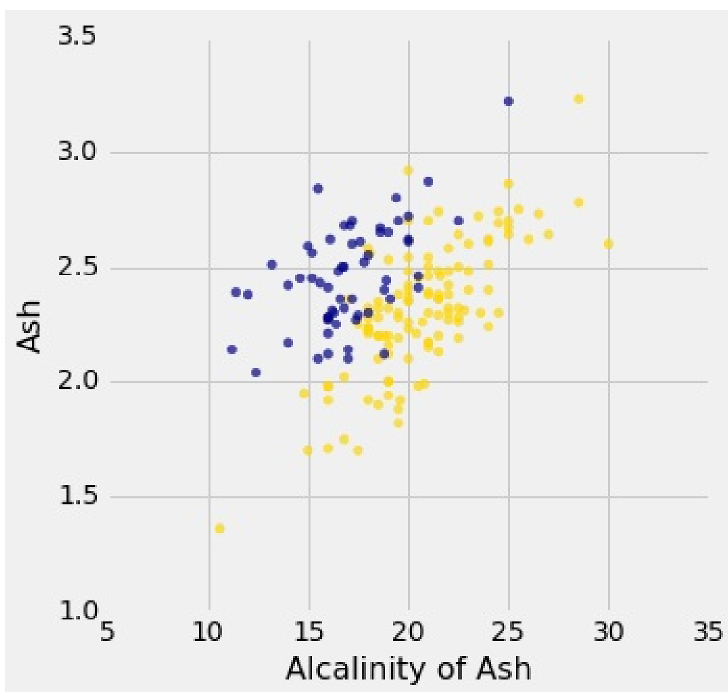
这也太大了！ 让我们做一些可视化，看看第一类是否真的看起来不同于第零类。

```
wine_with_colors = wine.join('Class', color_table)
wine_with_colors.scatter('Flavanoids', 'Alcohol', colors='Color')
```



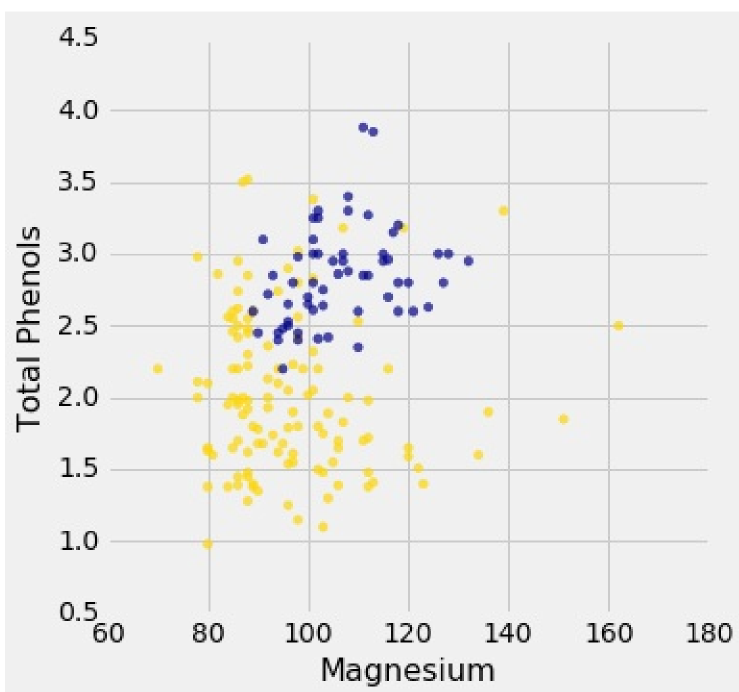
蓝点（第一类）几乎完全与金点分离。 这表明了，为什么两种第一类葡萄酒之间的距离小于两个不同类别葡萄酒之间的距离。 我们使用不同的一对属性，也可以看到类似的现象：

```
wine_with_colors.scatter('Alcalinity of Ash', 'Ash', colors='Color')
```



但是对于不同的偶对，图像更加模糊。

```
wine_with_colors.scatter('Magnesium', 'Total Phenols', colors='Color')
```



让我们来看看，是否可以基于所有的属性来实现一个分类器。 之后，我们会看到它有多准确。

实现计划

现在是时候编写一些代码来实现分类器了。 输入是我们要分类的一个点。 分类器的原理是，找到训练集中的 K 个最近邻点。 所以，我们的方法将会是这样：

找出最接近的 K 个点，即训练集中与点最相似的 K 个葡萄酒。

看看这些 K 个邻居的类别，并取大多数，找到最普遍的葡萄酒类别。 用它作为我们对点的预测。

所以这将指导我们的 Python 代码的结构。

```
def closest(training, p, k):
    ...

def majority(topkclasses):
    ...

def classify(training, p, k):
    kclosest = closest(training, p, k)
    kclosest.classes = kclosest.select('Class')
    return majority(kclosest)
```

实现步骤 1

为了为肾病数据实现第一步，我们必须计算点到训练集中每个患者的距离，按照距离排序，并取出训练集中最接近的 K 个患者。

这就是我们在上一节中使用对应 Alice 的点所做的事情。 我们来概括一下这个代码。 我们将在这里重新定义 `distance` ，只是为了方便。

```
def distance(point1, point2):
    """Returns the distance between point1 and point2
    where each argument is an array
    consisting of the coordinates of the point"""
    return np.sqrt(np.sum((point1 - point2)**2))

def all_distances(training, new_point):
    """Returns an array of distances
    between each point in the training set
    and the new point (which is a row of attributes)"""
    attributes = training.drop('Class')
    def distance_from_point(row):
        return distance(np.array(new_point), np.array(row))
    return attributes.apply(distance_from_point)

def table_with_distances(training, new_point):
    """Augments the training table
    with a column of distances from new_point"""
    return training.with_column('Distance', all_distances(training, new_point))

def closest(training, new_point, k):
    """Returns a table of the k rows of the augmented table
    corresponding to the k smallest distances"""
    with_dists = table_with_distances(training, new_point)
    sorted_by_distance = with_dists.sort('Distance')
    topk = sorted_by_distance.take(np.arange(k))
    return topk
```

让我们看看它如何在我们的葡萄酒数据上工作。 我们只要取第一个葡萄酒，在所有葡萄酒中找到最近的五个邻居。 请记住，由于这个葡萄酒是数据集的一部分，因此它自己是最近的邻居。 所以我们应该预计看到，它在列表顶端，后面是其他四个。

首先让我们来提取它的属性：

```
special_wine = wine.drop('Class').row(0)
```

现在让我们找到它的五个最近邻：

```
closest(wine, special_wine, 5)
```

Class	Alcohol	Malic Acid	Ash	Alcalinity of Ash	Magnesium	Total Phenols	Flavanc
1	14.23	1.71	2.43	15.6	127	2.8	3.06
1	13.74	1.67	2.25	16.4	118	2.6	2.9
1	14.21	4.04	2.44	18.9	111	2.85	2.65
1	14.1	2.02	2.4	18.8	103	2.75	2.92
1	14.38	3.59	2.28	16	102	3.25	3.17

好的！ 第一行是最近邻，这是它自己 - `Distance` 中值为零，和预期一样。 所有五个最近邻都属于第一类，这与我们先前的观察结果一致，即第一类葡萄酒集中在某些维度。

实现步骤 2 和 3

接下来，我们需要获取最近邻的“最大计数”，并把我们的点分配给大多数的相同类别。

```
def majority(topkclasses):
    ones = topkclasses.where('Class', are.equal_to(1)).num_rows
    zeros = topkclasses.where('Class', are.equal_to(0)).num_rows
    if ones > zeros:
        return 1
    else:
        return 0

def classify(training, new_point, k):
    closestk = closest(training, new_point, k)
    topkclasses = closestk.select('Class')
    return majority(topkclasses)
classify(wine, special_wine, 5)
1
```

如果将 `special_wine` 改为数据集中的最后一个，我们的分类器是否能够判断它在第零类中嘛？

```
special_wine = wine.drop('Class').row(177)
classify(wine, special_wine, 5)
0
```

是的! 分类器弄对了。

但是我们还不知道它对于所有其它葡萄酒如何，而且无论如何我们都知道，测试已经属于训练集的葡萄酒可能过于乐观了。 在本章的最后部分，我们将葡萄酒分为训练集和测试集，然后测量分类器在测试集上的准确性。

分类器的准确性

为了看看我们的分类器做得如何，我们可以将 50% 的数据放入训练集，另外 50% 放入测试集。基本上，我们保留一些数据以便以后使用，所以我们可以用它来测量分类器的准确性。我们始终将这个称为测试集。有时候，人们会把你留下用于测试的数据叫做保留集，他们会把这个估计准确率的策略称为保留方法。

请注意，这种方法需要严格的纪律。在开始使用机器学习方法之前，你必须先取出一些数据，然后放在一边用于测试。你必须避免使用测试集来开发你的分类器：你不应该用它来帮助训练你的分类器或者调整它的设置，或者用头脑风暴的方式来改进你的分类器。相反，在最后你已经完成分类器之后，当你想要它的准确率的无偏估计时，你应该仅仅使用它使用一次。

测量我们的葡萄酒分类器的准确率

好吧，让我们应用保留方法来评估 K 最近邻分类器识别葡萄酒的有效性。数据集有 178 个葡萄酒，所以我们将随机排列数据集，并将其中的 89 个放在训练集中，其余 89 个放在测试集中。

```
shuffled_wine = wine.sample(with_replacement=False)
training_set = shuffled_wine.take(np.arange(89))
test_set  = shuffled_wine.take(np.arange(89, 178))
```

我们将使用训练集中的 89 个葡萄酒来训练分类器，并评估其在测试集上的表现。 为了让我们更轻松，我们将编写一个函数，在测试集中每个葡萄酒上评估分类器：

```
def count_zero(array):
    """Counts the number of 0's in an array"""
    return len(array) - np.count_nonzero(array)

def count_equal(array1, array2):
    """Takes two numerical arrays of equal length
    and counts the indices where the two are equal"""
    return count_zero(array1 - array2)

def evaluate_accuracy(training, test, k):
    test_attributes = test.drop('Class')
    def classify_testrow(row):
        return classify(training, row, k)
    c = test_attributes.apply(classify_testrow)
    return count_equal(c, test.column('Class')) / test.num_rows
```

现在到了答案揭晓的时候了，我们来看看我们做得如何。 我们将任意使用 `k = 5` 。

```
evaluate_accuracy(training_set, test_set, 5)
0.9213483146067416
```

对于一个简单的分类器来说，这个准确率完全不差。

乳腺癌诊断

现在我想展示乳腺癌诊断的例子。我受到布列塔尼·温格（Brittany Wenger）的启发，他在 2012 年赢得了谷歌科学竞赛，还是一位 17 岁的高中生。这是布列塔尼：

布列塔尼的科学竞赛项目是构建一个诊断乳腺癌的分类算法。由于她构建了一个精度接近 99% 的算法，她获得了大奖。

让我们看看我们能做得如何，使用我们在这个课程中学到的思路。

所以，让我告诉你一些数据集的信息。基本上，如果一个女性的乳房存在肿块，医生可能想要进行活检，看看它是否是癌症。有几个不同的过程用于实现它。布列塔尼专注于细针抽吸（FNA），因为它比替代方案的侵袭性小。医生得到一块样本，放在显微镜下，拍摄一张照

片，一个训练有素的实验室技术人员分析图像，来确定是否是癌症。我们得到一张图片，像下面这样：

不幸的是，区分良性和恶性可能是棘手的。因此，研究人员已经研究了机器学习的用法，来帮助完成这项任务。我们的想法是，我们要求实验室技术人员分析图像并计算各种属性：诸如细胞的通常大小，细胞大小之间有多少变化等等。然后，我们将尝试使用这些信息来预测（分类）样本是否是恶性的。我们有一套来自女性的过去样本的训练集，其中正确的诊断已知，我们希望我们的机器学习算法可以使用它们来学习如何预测未来样本的诊断。

我们最后得到了以下数据集。对于 `Class` 列，1 表示恶性（癌症）；0 意味着良性（不是癌症）。

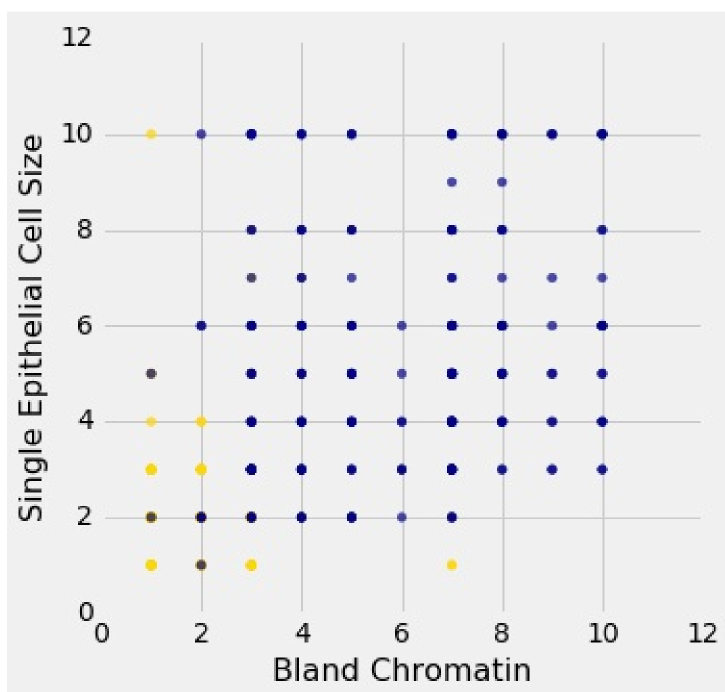
```
patients = Table.read_table('breast-cancer.csv').drop('ID')
patients
```

Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Blan Chrom
5	1	1	1	2	1	3
5	4	4	5	7	10	3
3	1	1	1	2	2	3
6	8	8	1	3	4	3
4	1	1	3	2	1	3
8	10	10	8	7	10	9
1	1	1	1	2	10	3
2	1	2	1	2	1	3
2	1	1	1	2	1	1
4	2	1	1	2	1	2

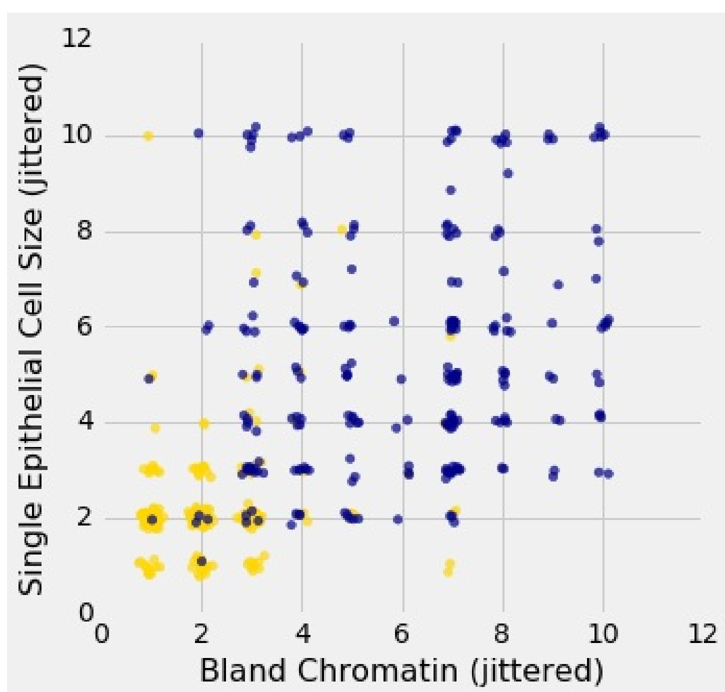
（省略了 673 行）

所以我们有 9 个不同的属性。 我不知道如何制作它们全部的 9 维散点图，所以我要挑选两个并绘制它们：

```
color_table = Table().with_columns(
    'Class', make_array(1, 0),
    'Color', make_array('darkblue', 'gold')
)
patients_with_colors = patients.join('Class', color_table)
patients_with_colors.scatter('Bland Chromatin', 'Single Epithelial Cell Size', colors=
'Color')
```



这个绘图完全是误导性的，因为有一堆点的 `x` 坐标和 `y` 坐标都有相同的值。 为了更容易看到所有的数据点，我将为 `x` 和 `y` 值添加一点点随机抖动。 这是看起来的样子：



例如，你可以看到有大量的染色质为 2 和上皮细胞大小为 2 的样本；所有都不是癌症。

请记住，抖动仅用于可视化目的，为了更容易感知数据。 我们现在已经准备好使用这些数据了，我们将使用原始数据（没有抖动）。

首先，我们将创建一个训练集和一个测试集。 数据集有 683 名患者，因此我们将随机排列数据集，并将其中的 342 个放在训练集中，其余的 341 个放在测试集中。

```
shuffled_patients = patients.sample(683, with_replacement=False)
training_set = shuffled_patients.take(np.arange(342))
test_set  = shuffled_patients.take(np.arange(342, 683))
```

让我们选取 5 个最近邻，并观察我们的分类器如何。

```
evaluate_accuracy(training_set, test_set, 5)
0.967741935483871
```

准确性超过 96%。不错！这样一个简单的技术再一次相当不错。

作为脚注，你可能已经注意到布列塔尼·温格做得更好了。 她使用了什么技术？ 一个关键的创新是，她将置信评分纳入了结果：她的算法有一种方法来确定何时无法做出有把握的预测，对于那些患者，甚至不尝试预测他们的诊断。 她的算法对于做出预测的病人是 99% 准确的，所以这个扩展看起来有点帮助。

多元回归

现在我们已经探索了使用多个属性来预测类别变量的方法，让我们返回来预测定量变量。 预测数值量被称为回归，多个属性进行回归的常用方法称为多元线性回归。

房价

下面的房价和属性数据集在爱荷华州埃姆斯市收集了数年。 数据集的描述在线显示。 我们将仅仅关注列的一个子集。 我们将尝试从其它列中预测价格列。

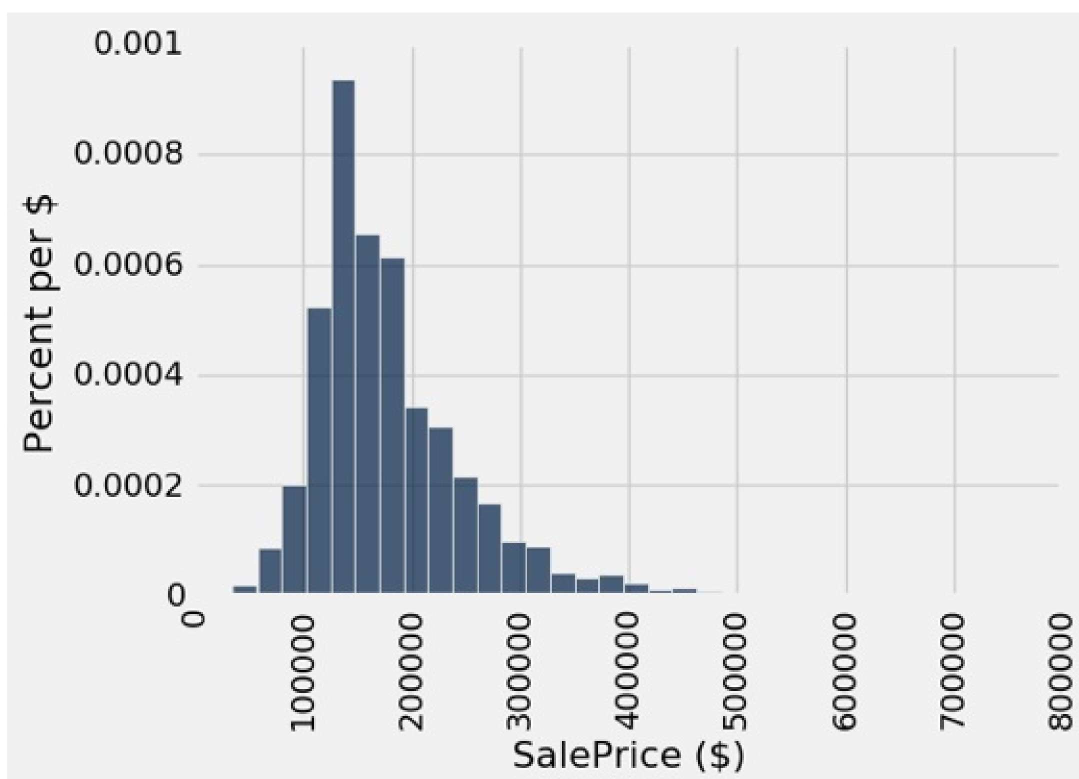
```
all_sales = Table.read_table('house.csv')
sales = all_sales.where('Bldg Type', '1Fam').where('Sale Condition', 'Normal').select(
    'SalePrice', '1st Flr SF', '2nd Flr SF',
    'Total Bsmt SF', 'Garage Area',
    'Wood Deck SF', 'Open Porch SF', 'Lot Area',
    'Year Built', 'Yr Sold')
sales.sort('SalePrice')
```

SalePrice	1st Flr SF	2nd Flr SF	Total Bsmt SF	Garage Area	Wood Deck SF	Open Porch SF	Lot Area	Year Built
35000	498	0	498	216	0	0	8088	1922
39300	334	0	0	0	0	0	5000	1946
40000	649	668	649	250	0	54	8500	1920
45000	612	0	0	308	0	0	5925	1940
52000	729	0	270	0	0	0	4130	1935
52500	693	0	693	0	0	20	4118	1941
55000	723	363	723	400	0	24	11340	1920
55000	796	0	796	0	0	0	3636	1922
57625	810	0	0	280	119	24	21780	1910
58500	864	0	864	200	0	0	8212	1914

（省略了 1992 行）

销售价格的直方图显示出大量的变化，分布显然不是正态。 右边的长尾包含几个价格非常高的房屋。 左边的短尾不包含任何售价低于 35,000 美元的房屋。

```
sales.hist('SalePrice', bins=32, unit='$')
```

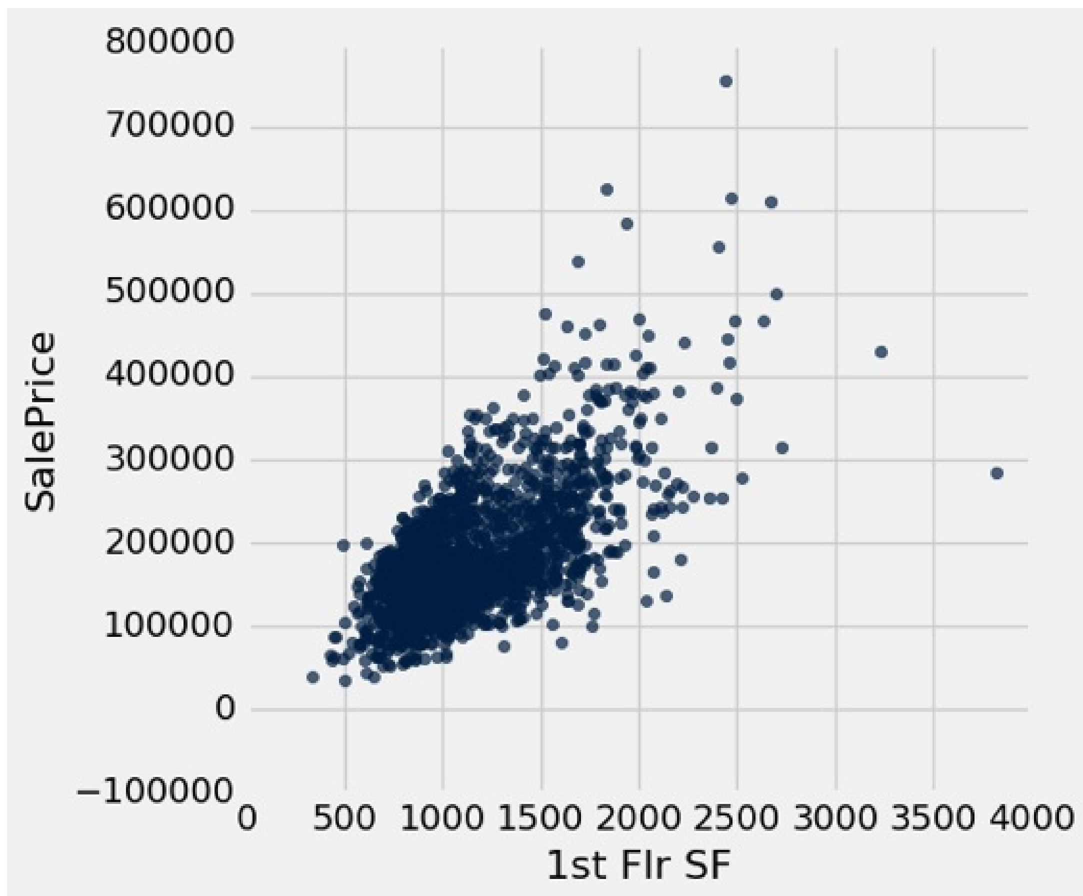


相关性

没有单个属性足以预测销售价格。例如，第一层面积（平方英尺）与销售价格相关，但仅解释其一些变化。

```
sales.scatter('1st Flr SF', 'SalePrice')

correlation(sales, 'SalePrice', '1st Flr SF')
0.64246625410302249
```



事实上，没有任何单个属性与销售价格的相关性大于 0.7（销售价格本身除外）。

```
for label in sales.labels:
    print('Correlation of', label, 'and SalePrice:\t', correlation(sales, label, 'Sale
Price'))
Correlation of SalePrice and SalePrice:     1.0
Correlation of 1st Flr SF and SalePrice:     0.642466254103
Correlation of 2nd Flr SF and SalePrice:     0.35752189428
Correlation of Total Bsmt SF and SalePrice:     0.652978626757
Correlation of Garage Area and SalePrice:     0.638594485252
Correlation of Wood Deck SF and SalePrice:     0.352698666195
Correlation of Open Porch SF and SalePrice:     0.336909417026
Correlation of Lot Area and SalePrice:     0.290823455116
Correlation of Year Built and SalePrice:     0.565164753714
Correlation of Yr Sold and SalePrice:     0.0259485790807
```

但是，组合属性可以提供更高的相关性。特别是，如果我们总结一楼和二楼的面积，那么结果的相关性就比任何单独的属性都要高。

```
both_floors = sales.column(1) + sales.column(2)
correlation(sales.with_column('Both Floors', both_floors), 'SalePrice', 'Both Floors')
0.7821920556134877
```

这种高度相关性表明，我们应该尝试使用多个属性来预测销售价格。 在具有多个观测属性和要预测的单个数值（这里是销售价格）的数据集中，多重线性回归可能是有效的技术。

多元线性回归

在多元线性回归中，通过将每个属性值乘以不同的斜率，从数值输入属性预测数值输出，然后对结果求和。 在这个例子中，第一层的斜率将代表房屋第一层面积的美元每平方英尺，它应该用于我们的预测。

在开始预测之前，我们将数据随机分成一个相同大小的训练和测试集。

```
train, test = sales.split(1001)
print(train.num_rows, 'training and', test.num_rows, 'test instances.')
1001 training and 1001 test instances.
```

多元回归中的斜率是一个数组，例子中每个属性拥有一个斜率值。 预测销售价格包括，将每个属性乘以斜率并将结果相加。

```
def predict(slopes, row):
    return sum(slopes * np.array(row))

example_row = test.drop('SalePrice').row(0)
print('Predicting sale price for:', example_row)
example_slopes = np.random.normal(10, 1, len(example_row))
print('Using slopes:', example_slopes)
print('Result:', predict(example_slopes, example_row))
Predicting sale price for: Row(1st Flr SF=1092, 2nd Flr SF=1020, Total Bsmt SF=952.0,
Garage Area=576.0, Wood Deck SF=280, Open Porch SF=0, Lot Area=11075, Year Built=1969,
 Yr Sold=2008)
Using slopes: [  9.99777721   9.019661    11.13178317   9.40645585  11.07998556
  11.03830075  10.26908341  10.42534332  11.00103437]
Result: 195583.275784
```

结果是估计的销售价格，可以将其与实际销售价格进行比较，以评估斜率是否提供准确的预测。 由于上面的`example_slopes`是随机选取的，我们不应该期望它们提供准确的预测。

```
print('Actual sale price:', test.column('SalePrice').item(0))
print('Predicted sale price using random slopes:', predict(example_slopes, example_row
))
Actual sale price: 206900
Predicted sale price using random slopes: 195583.275784
```

最小二乘回归

执行多元回归的下一步是定义最小二乘目标。 我们对训练集中的每一行执行预测，然后根据实际价格计算预测的均方根误差（RMSE）。

```
train_prices = train.column(0)
train_attributes = train.drop(0)

def rmse(slopes, attributes, prices):
    errors = []
    for i in np.arange(len(prices)):
        predicted = predict(slopes, attributes.row(i))
        actual = prices.item(i)
        errors.append((predicted - actual) ** 2)
    return np.mean(errors) ** 0.5

def rmse_train(slopes):
    return rmse(slopes, train_attributes, train_prices)

print('RMSE of all training examples using random slopes:', rmse_train(example_slopes)
)
RMSE of all training examples using random slopes: 69653.9880638
```

最后，我们使用 `minimize` 函数来找到使 RMSE 最低的斜率。 由于我们想要最小化的函数 `rmse_train` 需要一个数组而不是一个数字，所以我们必须向 `minimize` 函数传递 `array = True` 参数。 当使用这个参数时，`minimize` 也需要斜率的初始猜测，以便知道输入数组的维数。 最后，为了加速优化，我们使用 `smooth = True` 属性，指出 `rmse_train` 是一个平滑函数。 计算最佳斜率可能需要几分钟的时间。

```
best_slopes = minimize(rmse_train, start=example_slopes, smooth=True, array=True)
print('The best slopes for the training set:')
Table(train_attributes.labels).with_row(list(best_slopes)).show()
print('RMSE of all training examples using the best slopes:', rmse_train(best_slopes))
The best slopes for the training set:
```

1st Flr SF	2nd Flr SF	Total Bsmt SF	Garage Area	Wood Deck SF	Open Porch SF	Lot Area	Year Built
73.7779	72.3057	51.8885	46.5581	39.3267	11.996	0.451265	538.24

```
RMSE of all training examples using the best slopes: 31146.4442711
```

解释多元线性回归

让我们来解释这些结果。 最佳斜率为我们提供了一个方法，从其房屋属性估算价格。 一楼的面积约为 75 美元每平方英尺（第一个斜率），而二楼的面积约为 70 元每平方英尺（第二个斜率）。 最后的负值描述了市场：最近几年的价格平均较低。

大约 3 万美元的 RMSE 意味着，我们基于所有属性的销售价格的最佳线性预测，在训练集上平均差了大约 3 万美元。 当预测测试集的价格时，我们发现了类似的误差，这表明我们的预测方法可推广到来自同一总体的其他样本。

```
test_prices = test.column(0)
test_attributes = test.drop(0)

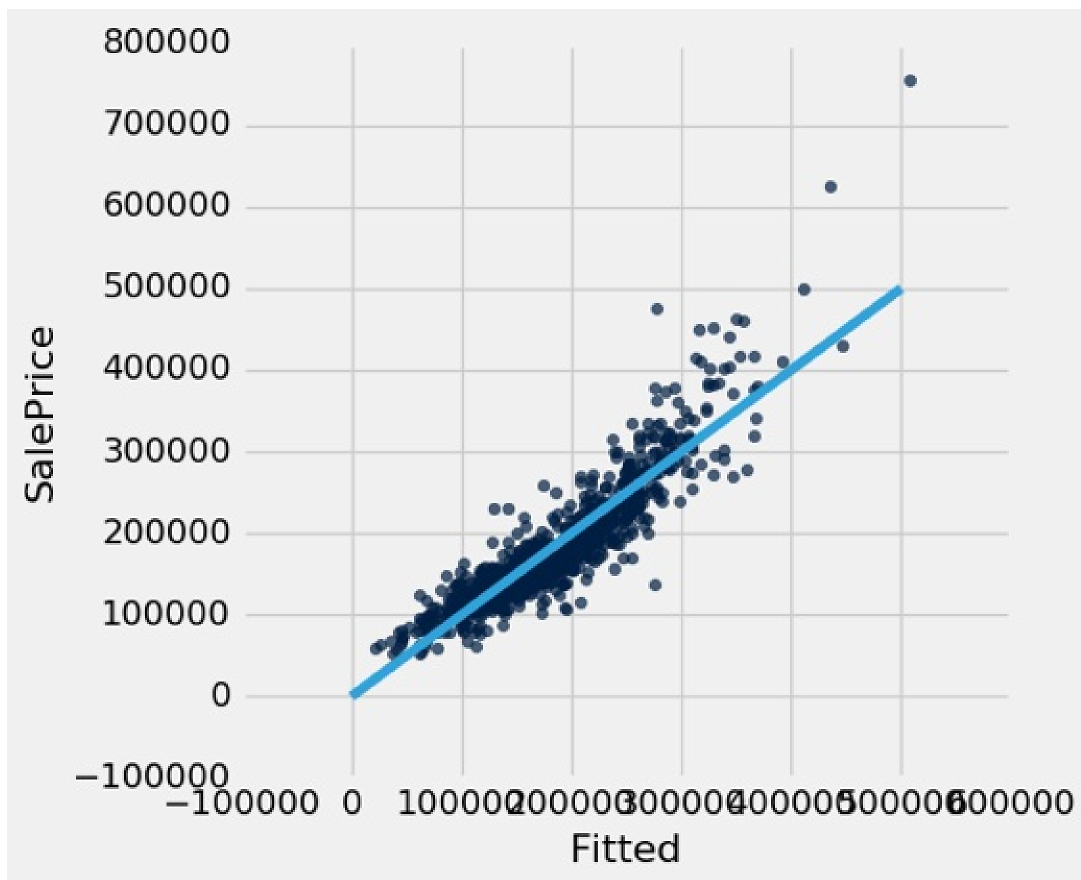
def rmse_test(slopes):
    return rmse(slopes, test_attributes, test_prices)

rmse_linear = rmse_test(best_slopes)
print('Test set RMSE for multiple linear regression:', rmse_linear)
Test set RMSE for multiple linear regression: 31105.4799398
```

如果预测是完美的，那么预测值和实际值的散点图将是一条斜率为 1 的直线。我们可以看到大多数点落在该线附近，但预测中存在一些误差。

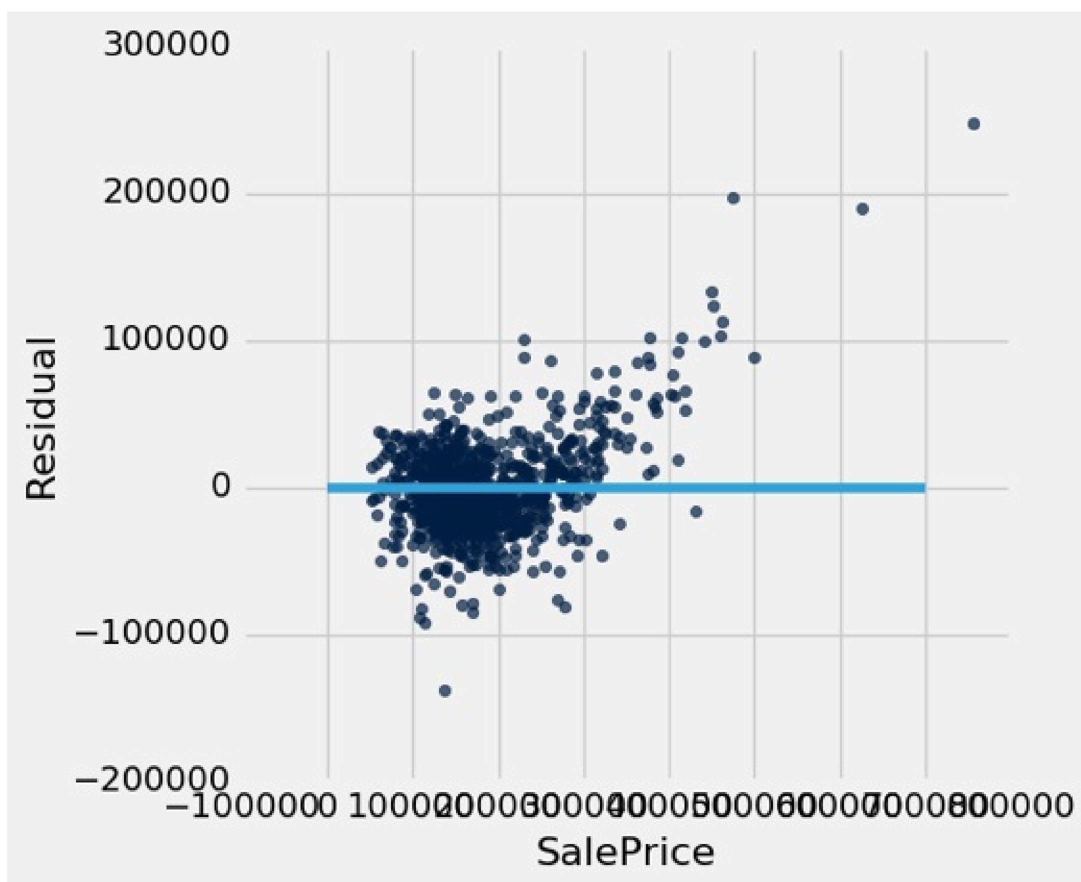
```
def fit(row):
    return sum(best_slopes * np.array(row))

test.with_column('Fitted', test.drop(0).apply(fit)).scatter('Fitted', 0)
plots.plot([0, 5e5], [0, 5e5]);
```



多元回归的残差图通常将误差（残差）与预测变量的实际值进行比较。 我们在下面的残差图中看到，我们系统性低估了昂贵房屋的值，由图右侧的许多正的残差值所示。

```
test.with_column('Residual', test_prices-test.drop(0).apply(fit)).scatter(0, 'Residual'
)
plots.plot([0, 7e5], [0, 0]);
```



就像简单的线性回归一样，解释预测结果至少和预测一样重要。 很多解释多元回归的课程不包含在这个课本中。 完成这门课之后的下一步自然是深入研究线性建模和回归。

最近邻回归

另一种预测房屋销售价格的方法是使用类似房屋的价格。 这个最近邻的方法与我们的分类器非常相似。 为了加速计算，我们将只使用与原始分析中销售价格相关性最高的属性。

```
train_nn = train.select(0, 1, 2, 3, 4, 8)
test_nn = test.select(0, 1, 2, 3, 4, 8)
train_nn.show(3)
```

SalePrice	1st Flr SF	2nd Flr SF	Total Bsmt SF	Garage Area	Year Built
240000	1710	0	1710	550	2004
229000	1302	735	672	472	1996
136500	864	0	864	336	1978

（省略了 998 行）

最近邻的计算与最近邻分类器相同。 在这种情况下，我们将从距离计算中排除 `'SalePrice'` 而不是 `'Class'` 列。 第一个测试行的五个最近邻如下所示。

```
def distance(pt1, pt2):
    """The distance between two points, represented as arrays."""
    return np.sqrt(sum((pt1 - pt2) ** 2))

def row_distance(row1, row2):
    """The distance between two rows of a table."""
    return distance(np.array(row1), np.array(row2))

def distances(training, example, output):
    """Compute the distance from example for each row in training."""
    dists = []
    attributes = training.drop(output)
    for row in attributes.rows:
        dists.append(row_distance(row, example))
    return training.with_column('Distance', dists)

def closest(training, example, k, output):
    """Return a table of the k closest neighbors to example."""
    return distances(training, example, output).sort('Distance').take(np.arange(k))

example_nn_row = test_nn.drop(0).row(0)
closest(train_nn, example_nn_row, 5, 'SalePrice')
```

SalePrice	1st Flr SF	2nd Flr SF	Total Bsmt SF	Garage Area	Year Built	Distance
150000	1299	0	967	494	1954	51.9711
144000	1344	0	1024	484	1958	60.8358
183500	1299	0	1001	486	1979	68.6003
140000	1283	0	931	506	1962	76.5049
173000	1287	0	957	541	1977	77.2464

预测价格的一个简单方法是计算最近邻的价格均值。

```
def predict_nn(example):
    """Return the majority class among the k nearest neighbors."""
    return np.average(closest(train_nn, example, 5, 'SalePrice').column('SalePrice'))

predict_nn(example_nn_row)
158100.0
```

最后，我们可以使用一个测试样本，检查我们的预测是否接近真实销售价格。 看起来很合理！

```
print('Actual sale price:', test_nn.column('SalePrice').item(0))
print('Predicted sale price using nearest neighbors:', predict_nn(example_nn_row))
Actual sale price: 146000
Predicted sale price using nearest neighbors: 158100.0
```

尾注

为了为整个测试集评估这个方法的性能，我们将 `predict_nn` 应用于每个测试示例，然后计算预测的均方根误差。 预测的计算可能需要几分钟的时间。

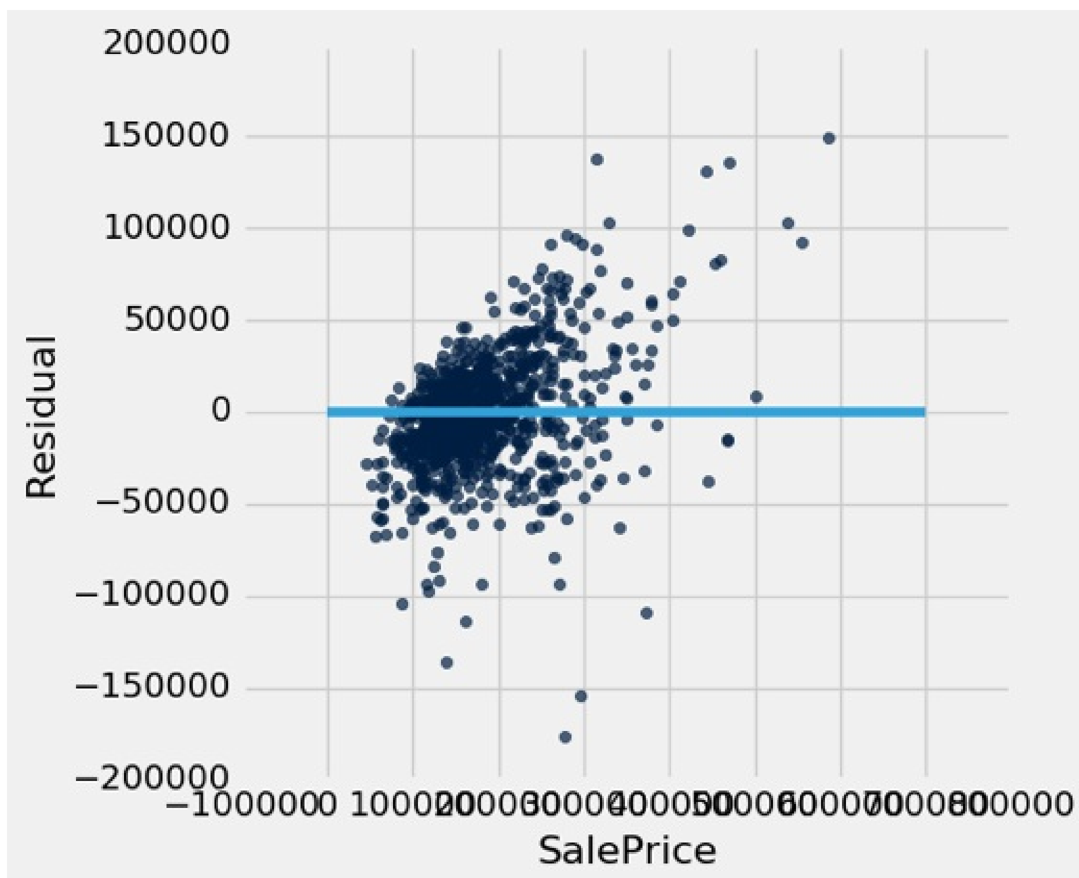
```
nn_test_predictions = test_nn.drop('SalePrice').apply(predict_nn)
rmse_nn = np.mean((test_prices - nn_test_predictions) ** 2) ** 0.5

print('Test set RMSE for multiple linear regression: ', rmse_linear)
print('Test set RMSE for nearest neighbor regression:', rmse_nn)
Test set RMSE for multiple linear regression:  30232.0744208
Test set RMSE for nearest neighbor regression: 31210.6572877
```

对于这些数据，这两种技术的误差非常相似！ 对于不同的数据集，一种技术可能会胜过另一种。 通过计算两种技术在同一数据上的均方根误差，我们可以公平比较这些方法。值得注意的是：表现的差异可能不完全由于技术；这可能由于随机变化，由于首先对训练和测试集进行抽样。

最后，我们可以为这些预测画出一个残差图。 我们仍然低估了最昂贵房屋的价格，但偏差似乎并不像系统性的。 然而，较低价格的残差非常接近零，这表明较低价格的预测准确性非常高。

```
test.with_column('Residual', test_prices-nn_test_predictions).scatter(0, 'Residual')
plots.plot([0, 7e5], [0, 0]);
```



十六、比较两个样本

> 原文：Comparing Two Samples
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

最近邻分类方法的动机是这样的，个体可能像最近的邻居。 从另一个角度来看，我们可以说一个类别的个体不像另一个类别中的个体。 机器学习为我们提供了一种有力的方法来发现这种相似性的缺乏，并将其用于分类。 它揭示了一种模式，通过一次检查一两个属性，我们不一定能发现它。

但是，我们可以从属性中学到很多东西。 为了了解它，我们将比较两个类中的属性分布。

让我们来看看 Brittany Wenger 的乳腺癌数据，看看是否只用一个属性，就有希望生成一个合理的分类器。 和以前一样，我们将在随机选择的训练集上进行探索，然后在剩余的保留集上测试我们的分类器。

```
patients = Table.read_table('breast-cancer.csv').drop('ID')
shuffled_patients = patients.sample(with_replacement=False)
training_set = shuffled_patients.take(np.arange(341))
test_set  = shuffled_patients.take(np.arange(341, 683))
training_set
```

Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Blar Chron
5	1	1	1	2	1	2
5	1	1	1	1	1	1
4	1	1	1	2	1	1
5	1	2	1	2	1	3
4	10	8	5	4	1	10
7	2	4	1	3	4	3
9	4	5	10	6	10	4
3	1	1	1	2	2	3
3	2	1	1	2	1	2
6	3	3	5	3	10	3

（省略了 331 行）

让我们看看第二个属性 `Uniformity of Cell Size`，能告诉我们患者分类的什么事情。

```
training_cellsize = training_set.select('Class', 'Uniformity of Cell Size').relabel(1,
'Uniformity')
training_cellsize
```

Class	Uniformity
0	1
0	1
0	1
0	1
1	10
1	2
1	4
0	1
0	2
0	3

（省略了 331 行）

`Class` 和 `Uniformity` 列显示为数字，但他们真的都是类别值。这些类别是“癌症”（1）和“非癌症”（0）。`Uniformity` 为 1-10，但是这些标签是由人确定的，他们也可能有十个标签，如“非常一致”，“不一致”等等。（一致性的 2 不一定是 1 的两倍。）所以我们比较两个类别分布，每个分类一个。

对于每个类别和每个一致评分，我们都需要训练集的患者数量。`pivot` 方法将为我们计数。

```
training_counts = training_cellsize.pivot('Class', 'Uniformity')
training_counts
```

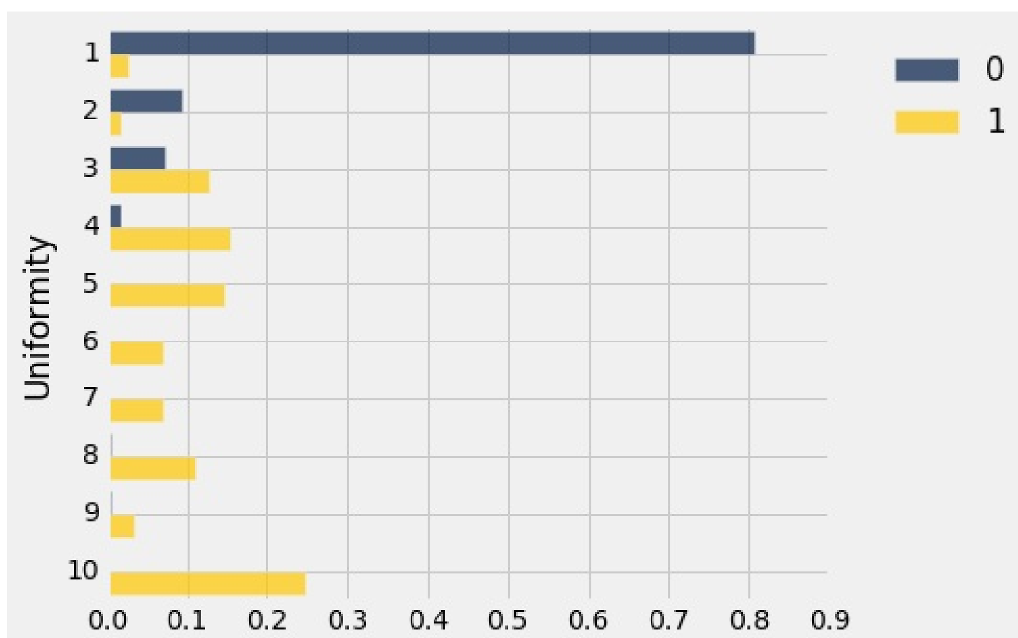
Uniformity	0	1
1	181	3
2	21	2
3	16	15
4	4	18
5	0	17
6	0	8
7	0	8
8	1	13
9	1	4
10	0	29

我们现在有了一些东西，类似于每个类别的一致评分的分布。 而这两者看起来相当不同。 但是，我们要小心 - 这两个类别的患者总数是 341（训练集的大小），超过一半的人在类别 0 里面。

```
np.sum(training_counts.column('0'))
224
```

所以为了比较两个分布，我们应该把计数转换成比例然后可视化。

```
def proportions(array):
    return array/np.sum(array)
training_dists = training_counts.select(0).with_columns(
   '0', proportions(training_counts.column('0')),
    '1', proportions(training_counts.column('1'))
)
training_dists.barh('Uniformity')
```



这两个分布看起来不一样！ 事实上，它们看起来相当不同，我们应该能够基于对这种差异的直截了当的观察来构建一个非常合理的分类器。 一个简单的分类规则是："如果一致性大于 3，类别就是 1，也就是说这个单元格就有癌症的，否则类别就是 0。

这么粗糙的东西有什么好处吗？ 让我们试试看。 对于测试集中的任何个体，我们所要做的就是，查看一致评分是否大于 3。例如，对于前 4 名患者，我们将得到一组四个布尔值：

```
test_set.take(np.arange(4)).column('Uniformity of Cell Size') > 3
array([ True, False, False, False], dtype=bool)
```

请记住， `True` 等于 `1` ，如果一致性大于 3，那么这是我们要划分的分类。因此，为了测量粗分类器的准确性，我们所要做的就是，求得测试集患者的比例， 其中分类与患者已知的分类相同。 我们将使用上一节中写的 `count_equal` 函数。

```
classification = test_set.column('Uniformity of Cell Size') > 3

count_equal(classification, test_set.column('Class'))/test_set.num_rows
0.935672514619883
```

这相当准确，即使我们只使用单个属性单行代码的分类器！

这是否意味着上一章中最近邻的方法是不必要的？ 不，因为那些更准确，并且对于癌症诊断，任何患者都想要尽可能精确的方法。 但是看到简单的方法并不坏，这是令人欣慰的。

两个类别分布

为了查看两个数值变量如何相关，可以使用相关系数来衡量线性关联。 但是，我们应该如何确定两个分类变量是否相关？ 例如，我们如何决定一个属性是否与个体的类别有关？ 这是一个很重要的问题，因为如果不相关的话，你可以把它从你的分类器中删除。

在乳腺癌数据中，我们来看看有丝分裂活动是否与这个类别有关。 我们已经标记了“癌症”和“非癌症”的类别，以便以后参考。

```
classes = Table().with_columns(
    'Class', make_array(0, 1),
    'Class Label', make_array('Not Cancer', 'Cancer')
)
patients = Table.read_table('breast-cancer.csv').drop('ID').join('Class', classes)
patients = patients.drop('Class').relabel('Class Label', 'Class')
mitoses = patients.select('Class', 'Mitoses')
mitoses
```

Class	Mitoses
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	5
Not Cancer	1
Not Cancer	1

（省略了 673 行）

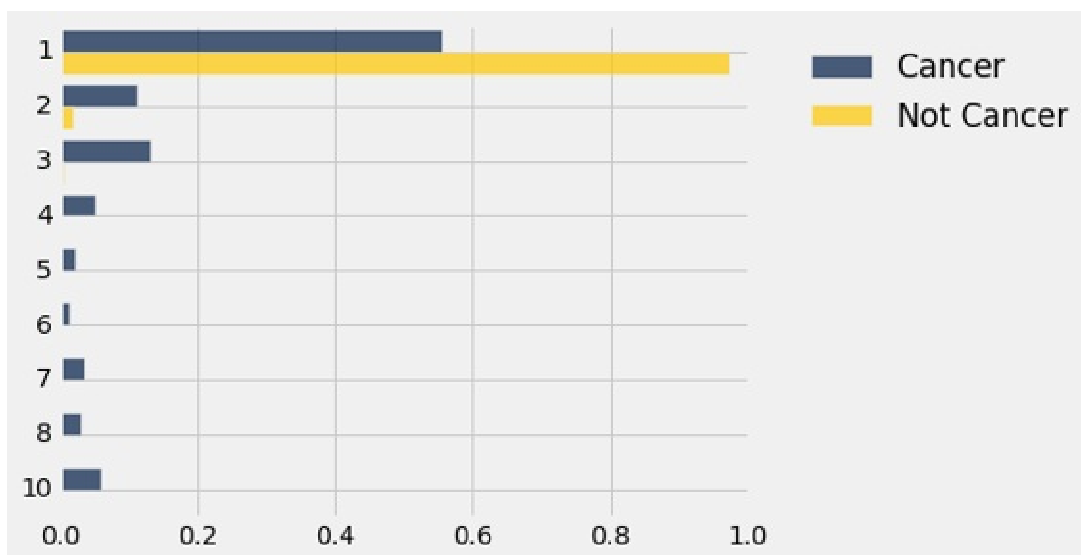
我们可以使用 `pivot` 和 `proportions` （在前面的章节中定义）来显示两类中 `Mitoses` 的分布。

```
counts = mitoses.pivot('Class', 'Mitoses')
counts
```

Mitoses	Cancer	Not Cancer
1	132	431
2	27	8
3	31	2
4	12	0
5	5	1
6	3	0
7	8	1
8	7	1
10	14	0

```
dists = counts.select(0).with_columns(
    'Cancer', proportions(counts.column(1)),
    'Not Cancer', proportions(counts.column(2))

)
dists.barh(0)
```



与“非癌症”类别的分布相比，“癌症”类别的 `Mitoses` 都集中于最低评分。

所以看起来类别和有丝分裂活动是相关的。 但是，这可能只是由于偶然嘛？

为了了解偶然来自哪里，请记住，数据就像是来自更大总体的随机样本 - 总体包含我们可能要分类的新个体。 可能在总体中，类别和有丝分裂是相互独立的，只是由于偶然与样本相关。

假设

我们试着通过对以下假设进行测试来回答这个问题。

原假设。 在总体中，类别和有丝分裂评分是相互独立的；换句话说，这两个类别的有丝分裂的分布是一样的。 由于偶然性，样本分布是不同的。

备选假说。 在总体中，类别和有丝分裂评分是相关的。

为了了解如何测试它，我们再看一下数据。

```
mitoses
```

Class	Mitoses
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	1
Not Cancer	5
Not Cancer	1
Not Cancer	1

（省略了 673 行）

随机排列

如果类别和有丝分裂评分是不相关的，那么 `Mitoses` 值出现的顺序并不重要，因为它们与类别的值无关，所有的重新排列应该是等可能的。 这与我们在分析足球 `Deflategate` 数据时采用的方法相同。

所以让我们将所有的 `Mitoses` 值整理到一个名为 `shuffled_mitoses` 的数组中。 你可以看到下面的第一项，但它包含 683 个项目，因为它是整个 `Mitoses` 列的排列（即重新排列）。

```
shuffled_mitoses = mitoses.select('Mitoses').sample(with_replacement=False).column(0)
shuffled_mitoses.item(0)
1
```

让我们扩展 `mitoses` 表，添加一列乱序的值。

```
mitoses = mitoses.with_column('Shuffled Mitoses', shuffled_mitoses)
mitoses
```

Class	Mitoses	Shuffled Mitoses
Not Cancer	1	1
Not Cancer	1	1
Not Cancer	1	1
Not Cancer	1	1
Not Cancer	1	7
Not Cancer	1	1
Not Cancer	1	1
Not Cancer	5	3
Not Cancer	1	1
Not Cancer	1	2

（省略了 673 行）

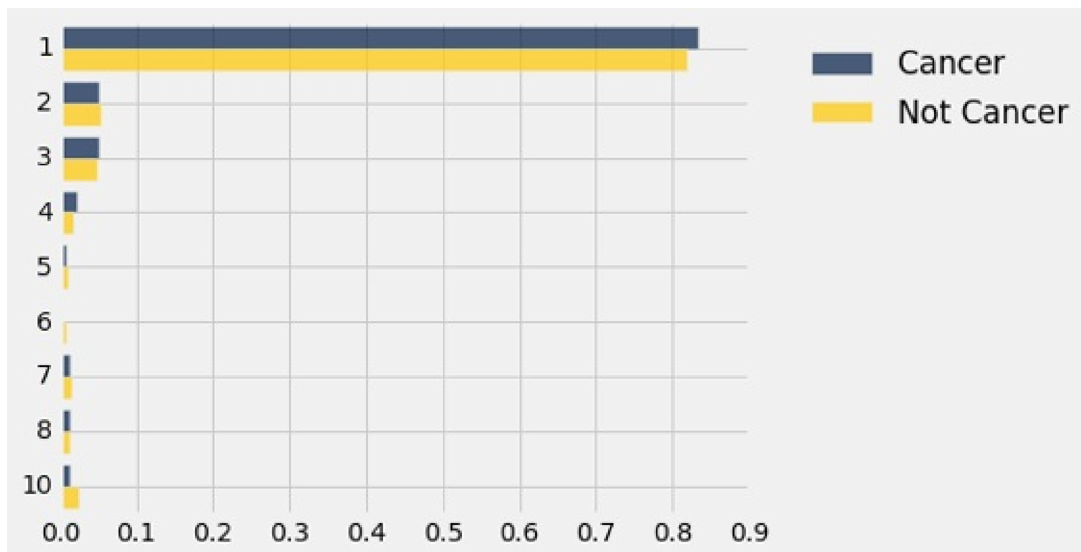
让我们看看乱序数据的有丝分裂的分布，使用与原始数据相同的过程。

```
shuffled = mitoses.select('Class', 'Shuffled Mitoses')
shuffled_counts = shuffled.pivot('Class', 'Shuffled Mitoses')
shuffled_counts
```

Shuffled Mitoses	Cancer	Not Cancer
1	199	364
2	12	23
3	12	21
4	5	7
5	2	4
6	0	3
7	3	6
8	3	5
10	3	11

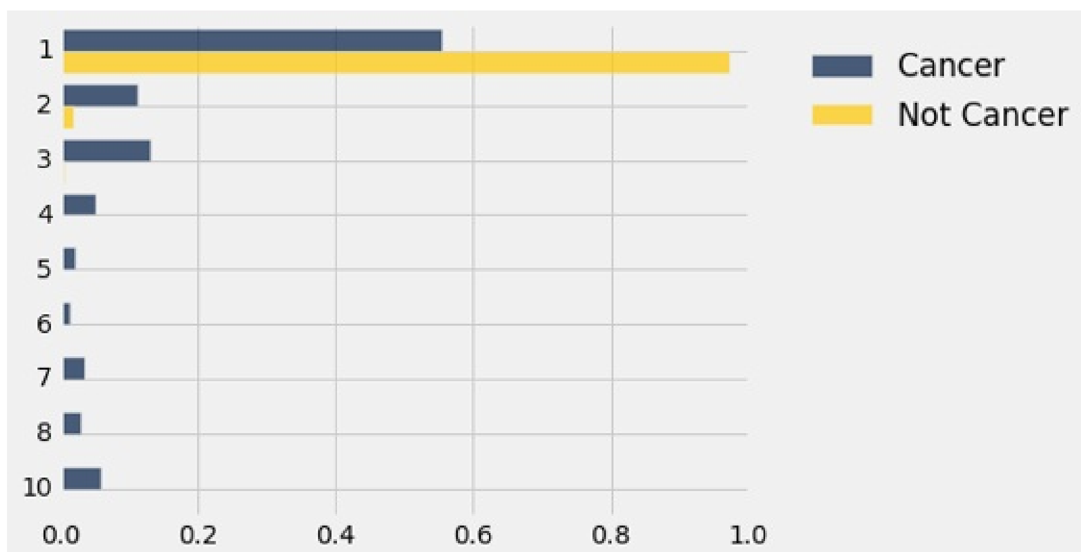
这两个类中的乱序数据的分布可以展示为条形图，就像原始数据一样。

```
shuffled_dists = shuffled_counts.select(0).with_columns(
    'Cancer', proportions(shuffled_counts.column(1)),
    'Not Cancer', proportions(shuffled_counts.column(2))
)
shuffled_dists.barh(0)
```



这与原始条形图看起来有点不同，为方便起见，再次展示如下。

```
dists.barh(0)
```



检验统计量：总变异距离

我们需要一个测试统计量来衡量蓝色和金色分布之间的差异。 回想一下，总变异距离可以用来量化两个类别分布的差异。

```
def tvd(dist1, dist2):
    return 0.5*(np.sum(np.abs(dist1 - dist2)))
```

在原始样本中，两个类别的有丝分裂的分布的 TVD 约为 0.4：

```
observed_tvd = tvd(dists.column(1), dists.column(2))
observed_tvd
0.41841946549059517
```

但是在乱序的样本中，它比较小：

```
tvd(shuffled_dists.column(1), shuffled_dists.column(2))
0.022173847487655045
```

随机排列的有丝分裂评分和原始评分似乎表现不一样。 但是如果我们再次运行，随机打乱可能会有所不同。 让我们重新打乱并重新计算总变异距离。

```
shuffled_mitoses = mitoses.select('Mitoses').sample(with_replacement=False).column(0)

shuffled = mitoses.select('Class').with_column('Shuffled Mitoses', shuffled_mitoses)

shuffled_counts = shuffled.pivot('Class', 'Shuffled Mitoses')

tvd(proportions(shuffled_counts.column(1)), proportions(shuffled_counts.column(2)))
0.039937426966715643
```

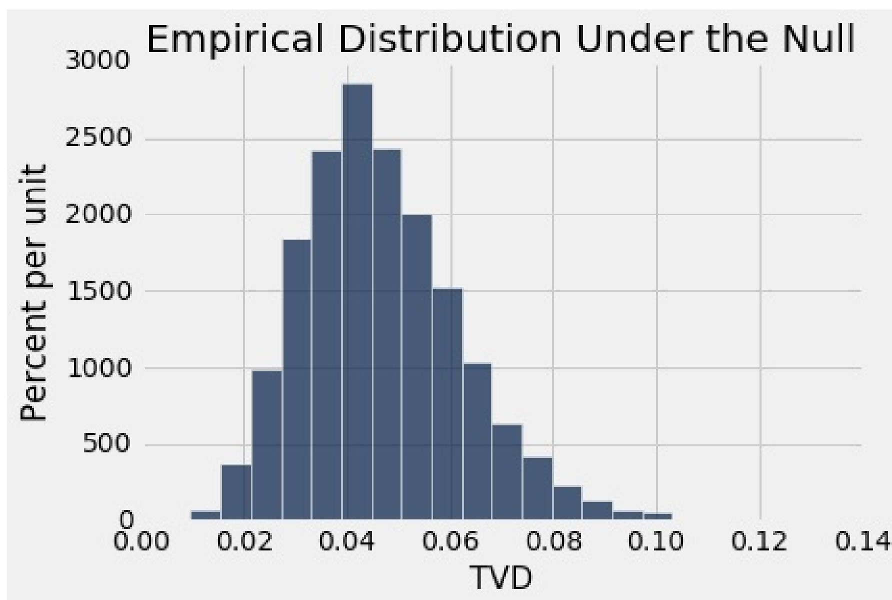
总变异距离仍然比我们从原始数据得到的 0.42 小很多。 为了看看它变化了多少，我们不得不重复多次随机打乱过程，在它现在已经变得很熟悉了。

原假设下 TVD 的经验分布

如果原假设是真的，则有丝分裂评分的所有排列都是等可能的。 有很多可能的排列；让我们做 5000 次，看看我们的检验统计量的变化。 代码与上面的代码完全一样，只是现在我们将收集所有 5000 个距离并绘制经验直方图。

```
repetitions = 5000
tvds = make_array()
for i in np.arange(repetitions):
    shuffled_mitoses = mitoses.select('Mitoses').sample(with_replacement=False).column(
0)
    shuffled = mitoses.select('Class').with_column('Shuffled Mitoses', shuffled_mitose
s)
    shuffled_counts = shuffled.pivot('Class', 'Shuffled Mitoses')
    new_tvd = tvd(proportions(shuffled_counts.column(1)), proportions(shuffled_counts.
column(2)))
    tvds = np.append(tvds, new_tvd)

Table().with_column('TVD', tvds).hist(bins=20)
plots.title('Empirical Distribution Under the Null')
print('Observed TVD:', observed_tvd)
Observed TVD: 0.418419465491
```



观察到的总变异距离 0.42 根本不接近于假设零假设为真所产生的分布。 数据支持备选假设：有丝分裂评分与类别有关。

两个类别分布的相等性的排列检验

我们上面所做的检验被称为原假设的排列检验，即两个样本是从相同的底层分布中抽取的。

为了定义一个执行检验的函数，我们可以复制前一个单元格的代码，并更改表和列的名称。函数 `permutation_test_tvd` 接受数据表的名称，包含类别变量的列标签，它的分布要检验，包含二元类别变量的列标签，以及要运行的随机排列的数量。

在我们上面的例子中，我们没有计算 P 值，因为观测值远离原假设下统计量的分布。但是，一般来说，我们应该计算 P 值，因为在其他例子中统计量可能不是那么极端。 P 值是“假设原假设为真，所得距离大于等于观测距离”的几率，因为备选假设比原假设预测了更大的距离。

```
def permutation_test_tvd(table, variable, classes, repetitions):

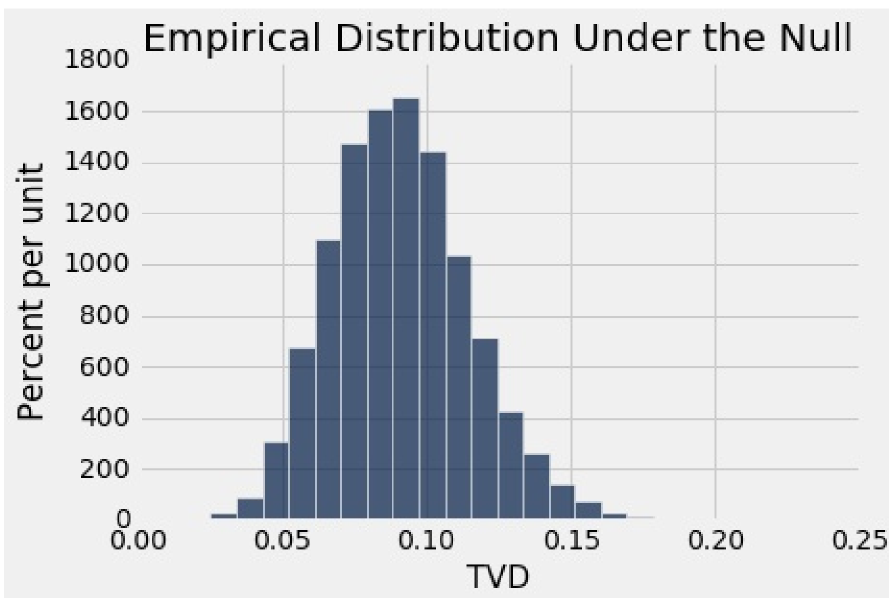
    """Test whether a categorical variable is independent of classes:
    table: name of table containing the sample
    variable: label of column containing categorical variable whose distribution is of
 interest
    classes: label of column containing binary class data
    repetitions: number of random permutations"""

    # Find the tvd between the distributions of variable in the two classes
    counts = table.select(classes, variable).pivot(classes, variable)
    observed_tvd = tvd(proportions(counts.column(1)), proportions(counts.column(2)))

    # Assuming the null is true, randomly permute the variable and collect all the new
 tvd's
    tvds = make_array()
    for i in np.arange(repetitions):
        shuffled_var = table.select(variable).sample(with_replacement=False).column(0)
        shuffled = table.select(classes).with_column('Shuffled Variable', shuffled_var
)
        shuffled_counts = shuffled.pivot(classes, 'Shuffled Variable')
        new_tvd =tvd(proportions(shuffled_counts.column(1)), proportions(shuffled_coun
ts.column(2)))
        tvds = np.append(tvds, new_tvd)

    # Find the empirical P-value:
    emp_p = np.count_nonzero(tvds >= observed_tvd)/repetitions

    # Draw the empirical histogram of the tvd's generated under the null,
    # and compare with the value observed in the original sample
    Table().with_column('TVD', tvds).hist(bins=20)
    plots.title('Empirical Distribution Under the Null')
    print('Observed TVD:', observed_tvd)
    print('Empirical P-value:', emp_p)
permutation_test_tvd(patients, 'Clump Thickness', 'Class', 5000)
Observed TVD: 0.638310905047
Empirical P-value: 0.0
```



同样，观测距离 0.64 离原假设预测的分布很远。 经验 P 值为 0，所以准确的 P 值将接近于零。 因此，如果类别和有丝分裂评分是不相关的，那么观测的数据是极不可能的。

所以得出的结论是，有丝分裂评分与类别有关，不仅在样本中，而且在总体中。

我们使用排列检验来帮助我们确定，类别属性的分布是否与类别相关。 一般来说，排列检验可以这样使用来确定，两个类别分布是否从相同的基本分布随机抽样。

A/B 测试

我们使用随机排列来查看，两个样本是否从相同的基本分类分布抽取。 如果样本是数值的，则可以使用相同的方法；检验统计量的选择通常比较简单。 在我们使用 `Deflategate` 数据的例子中，我们使用了不同的方法来测试爱国者队和小马队用球是否来自相同的基本分布。

在现代数据分析中，决定两个数值样本是否来自相同的基本分布称为 A/B 测试。 名称是指两个样本 A 和 B 的标签。

吸烟者和不吸烟者

我们对随机抽样的母亲及其新生儿进行了许多不同的分析，但是我们还没有查看母亲是否吸烟的数据。 研究的目的之一，是看母亲吸烟是否与出生体重有关。

```
baby = Table.read_table('baby.csv')
baby
```

Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Maternal Smoker
120	284	27	62	100	False
113	282	33	64	135	False
128	279	28	64	115	True
108	282	23	67	125	True
136	286	25	62	93	False
138	244	33	62	178	False
132	245	23	65	140	False
120	289	25	62	125	False
143	299	30	66	136	True
140	351	27	68	120	False

（省略了 1164 行）

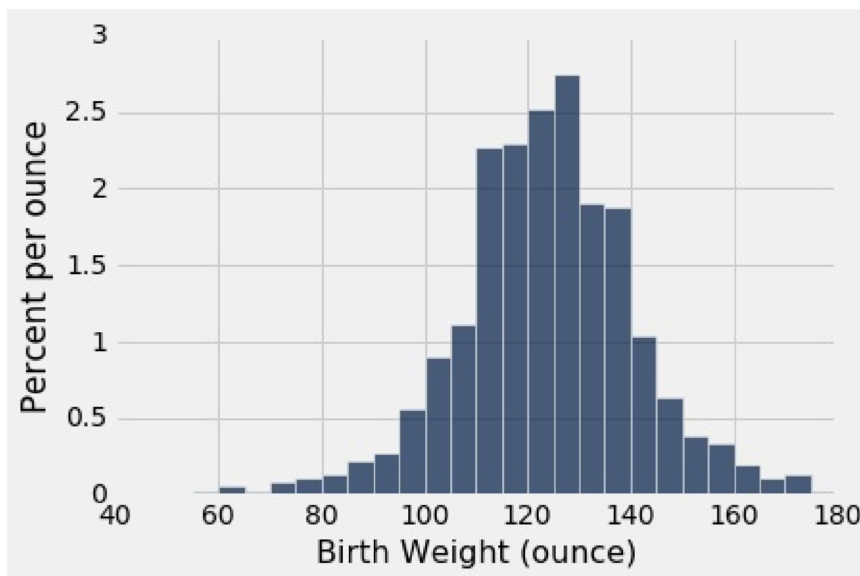
我们首先选择 `Birth Weight` 和 `Maternal Smoker` 。 样本中有 715 名非吸烟者，459 名吸烟者。

```
weight_smoke = baby.select('Birth Weight', 'Maternal Smoker')
weight_smoke.group('Maternal Smoker')
```

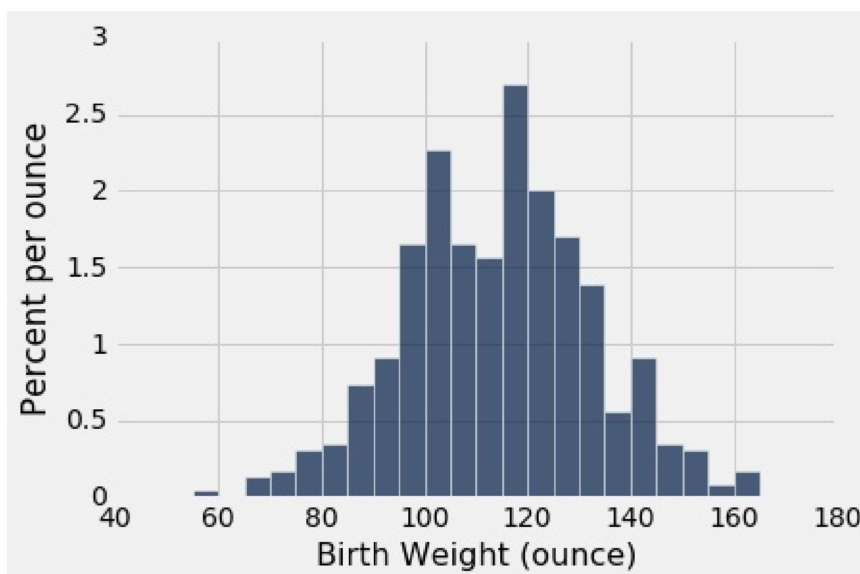
Maternal Smoker	count
False	715
True	459

下面的第一个直方图显示了样本中非吸烟者的婴儿出生体重的分布。 第二个显示了吸烟者的婴儿出生体重。

```
nonsmokers = baby.where('Maternal Smoker', are.equal_to(False))
nonsmokers.hist('Birth Weight', bins=np.arange(40, 181, 5), unit='ounce')
```




```
smokers = baby.where('Maternal Smoker', are.equal_to(True))
smokers.hist('Birth Weight', bins=np.arange(40, 181, 5), unit='ounce')
```



两种分布都大致是钟形，中心在 120 盎司附近。 当然，这些分布并不相同，这就产生了这样一个问题，即差异是否仅仅反映了机会变异，还是反映了总体分布的差异。

这个问题可以通过假设检验来回答。

原假设：在总体中，不吸烟的母亲的婴儿出生体重的分布和吸烟的母亲相同。 样本中的差异是偶然的。

备选假设：两种分布在总体中是不同的。

检验统计量：出生体重是一个定量变量，所以用均值的绝对差作为检验统计量是合理的。

检验统计量的观测值约为 9.27 盎司。

```
means_table = weight_smoke.group('Maternal Smoker', np.mean)
means_table
```

Maternal Smoker	Birth Weight mean
False	123.085
True	113.819

```
nonsmokers_mean = means_table.column(1).item(0)
smokers_mean = means_table.column(1).item(1)
nonsmokers_mean - smokers_mean
9.266142572024918
```

排列检验

为了看看原假设下是否有可能出现这种差异，我们将使用排列检验，就像我们在前一节中所做的那样。 我们必须为检验统计量改变代码。 为此，我们将像上面那样计算平均值的差，然后取绝对值。

请记住，在原假设下，出生体重的所有排列与 `Maternal Smoker` 列等可能出现。 所以，就像以前一样，每次重复都是打乱正在比较的变量。

```
def permutation_test_means(table, variable, classes, repetitions):

    """Test whether two numerical samples
    come from the same underlying distribution,
    using the absolute difference between the means.
    table: name of table containing the sample
    variable: label of column containing the numerical variable
    classes: label of column containing names of the two samples
    repetitions: number of random permutations"""

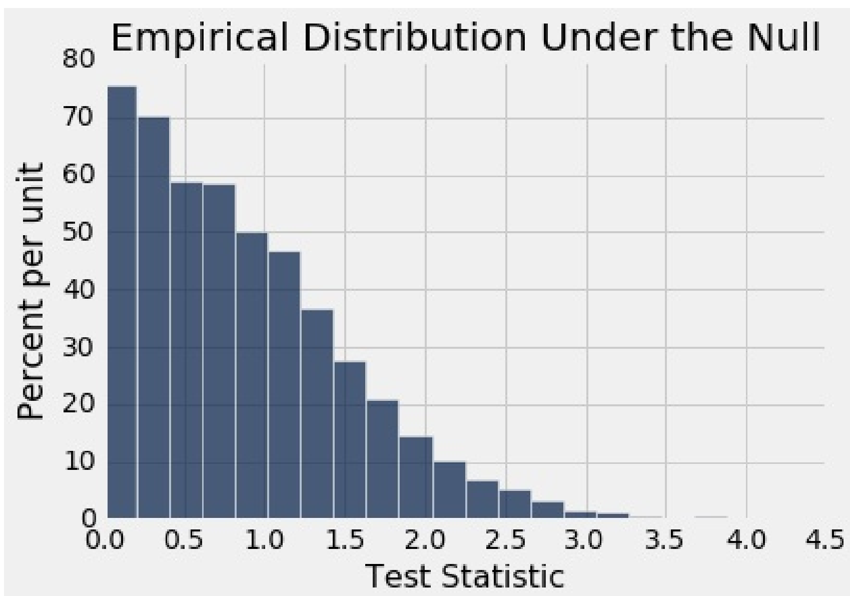
    t = table.select(variable, classes)

    # Find the observed test statistic
    means_table = t.group(classes, np.mean)
    obs_stat = abs(means_table.column(1).item(0) - means_table.column(1).item(1))

    # Assuming the null is true, randomly permute the variable
    # and collect all the generated test statistics
    stats = make_array()
    for i in np.arange(repetitions):
        shuffled_var = t.select(variable).sample(with_replacement=False).column(0)
        shuffled = t.select(classes).with_column('Shuffled Variable', shuffled_var)
        m_tbl = shuffled.group(classes, np.mean)
        new_stat = abs(m_tbl.column(1).item(0) - m_tbl.column(1).item(1))
        stats = np.append(stats, new_stat)

    # Find the empirical P-value:
    emp_p = np.count_nonzero(stats >= obs_stat)/repetitions

    # Draw the empirical histogram of the tvd's generated under the null,
    # and compare with the value observed in the original sample
    Table().with_column('Test Statistic', stats).hist(bins=20)
    plots.title('Empirical Distribution Under the Null')
    print('Observed statistic:', obs_stat)
    print('Empirical P-value:', emp_p)
permutation_test_means(baby, 'Birth Weight', 'Maternal Smoker', 5000)
Observed statistic: 9.266142572024918
Empirical P-value: 0.0
```



原始样本中的观测差异约为 9.27 盎司，与此分布不一致：经验 P 值为 0，这意味着确切的 P 值确实非常小。 因此，测试的结论是，在总体中，不吸烟者和吸烟者的婴儿出生体重的分布是不同的。

差值的自举置信区间

我们的 A/B 测试得出结论，这两个分布是不同的，但有点不尽人意。他们有多么不同？哪一个均值更大？这些自然是测试无法回答的问题。

回想一下，我们之前已经讨论过这个问题了：不仅仅是问“两个分布是否不同”的是与否的问题，我们可以通过不作任何假设，并简单地估计均值之间的差异，来学到更多。

观测差异（不吸烟者减去吸烟者）约为 9.27 盎司；这个正面迹象表明，不吸烟的母亲通常有更大的婴儿。但由于随机性，样本可能会有所不同。为了了解有多么不同，我们必须生成更多的样本；为了生成更多的样本，我们将使用 `bootstrap` ，就像我们以前做过的那样。自举过程不会假设这两个分布是否相同。它只是复制原始随机样本并计算统计量的新值。

函数 `bootstrap_ci_means` 返回总体中两组均值之间差异的自举置信区间。在我们的例子中，置信区间将估计总体中吸烟和不吸烟的母亲的婴儿的平均出生体重之间的差异。

- 表名称，它包含原始样本中的数据
- 列标签，它包含数值变量
- 列标签，它包含两个样本的名称
- 自举的重复次数

该函数使用自举百分比方法，返回两个均值之间的差异的约 95% 置信区间。

```
def bootstrap_ci_means(table, variable, classes, repetitions):

    """Bootstrap approximate 95% confidence interval
    for the difference between the means of the two classes
    in the population"""

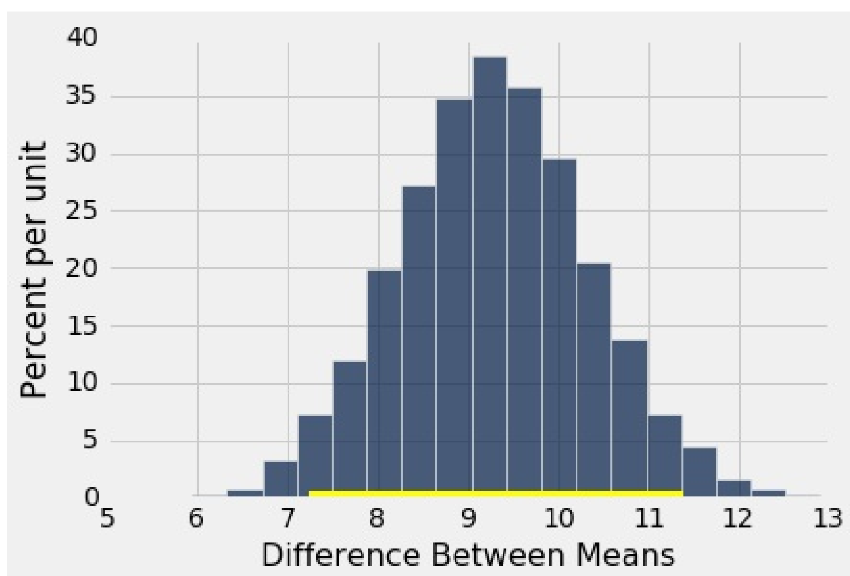
    t = table.select(variable, classes)

    mean_diffs = make_array()
    for i in np.arange(repetitions):
        bootstrap_sample = t.sample()
        m_tbl = bootstrap_sample.group(classes, np.mean)
        new_stat = m_tbl.column(1).item(0) - m_tbl.column(1).item(1)
        mean_diffs = np.append(mean_diffs, new_stat)

    left = percentile(2.5, mean_diffs)
    right = percentile(97.5, mean_diffs)

    # Find the observed test statistic
    means_table = t.group(classes, np.mean)
    obs_stat = means_table.column(1).item(0) - means_table.column(1).item(1)

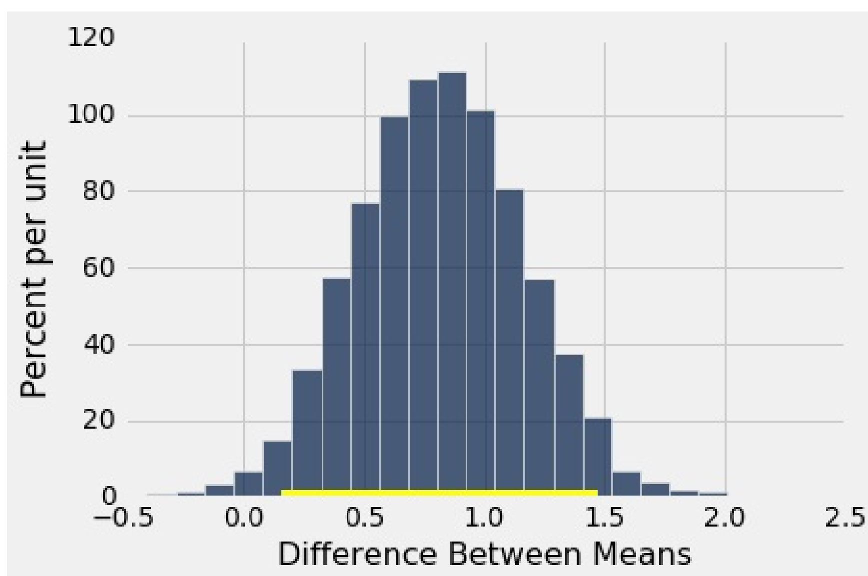
    Table().with_column('Difference Between Means', mean_diffs).hist(bins=20)
    plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=8)
    print('Observed difference between means:', obs_stat)
    print('Approximate 95% CI for the difference between means:')
    print(left, 'to', right)
bootstrap_ci_means(baby, 'Birth Weight', 'Maternal Smoker', 5000)
Observed difference between means: 9.266142572024918
Approximate 95% CI for the difference between means:
7.23940878698 to 11.3907887554
```



不吸烟的母亲的婴儿比吸烟的母亲的婴儿平均重 7.2 盎司到 11.4 盎司。 这比“两个分布不同”更有用。 由于置信区间不包含 0，它也告诉我们这两个分布是不同的。 所以置信区间估计了我们的均值之间的差异，也让我们决定两个基本分布是否相同。

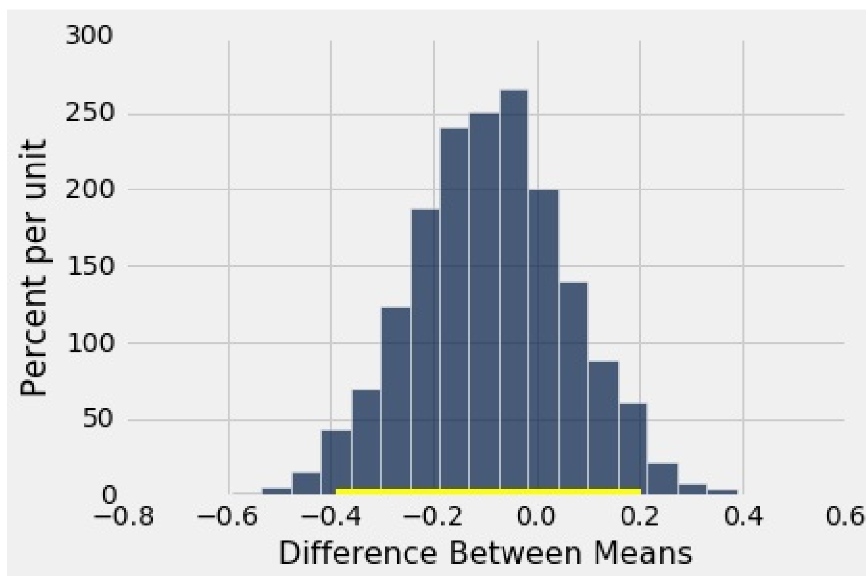
不吸烟的母亲比吸烟的母亲平均年龄稍大。

```
bootstrap_ci_means(baby, 'Maternal Age', 'Maternal Smoker', 5000)
Observed difference between means: 0.8076725017901509
Approximate 95% CI for the difference between means:
0.154278698588 to 1.4701157656
```



但毫不奇怪，证据并没有指出，他们的平均身高与不吸烟的母亲不同。 零在均值之间差异的置信区间中。

```
bootstrap_ci_means(baby, 'Maternal Height', 'Maternal Smoker', 5000)
Observed difference between means: 0.09058914941267915
Approximate 95% CI for the difference between means:
-0.390841928035 to 0.204388297872
```



总之：

如果你想知道两个基本分布是否相同，则可以使用带有适当检验统计量的排列检验。 当分布是类别时，我们使用总变异距离，而分布是数值时，我们使用均值之间的绝对差。

为了比较两个数值分布，将假设检验替换为估计，通常更富有信息。 只需估计一个差异，比如两组均值之间的差异。 这可以通过构建自举置信区间来完成。 如果零不在这个区间内，你可以得出这样的结论：这两个分布是不同的，你也可以估计均值有多么不同。

因果

我们用于比较两个样本的方法在随机对照实验的分析中具有强大的用途。由于在这些实验中，实验组和对照组被随机分配，因此如果实验完全没有效果，结果中的任何差异，可以与仅仅由于分配中的随机性而发生的情况进行比较。如果观察到的差异比我们预测的，纯粹由于偶然的差异更为显着，我们就会有因果关系的证据。由于个体无偏分配到实验组和对照组，两组结果中的差异可归因于实验。

随机对照实验分析的关键是，了解偶然因素如何出现。这有助于我们设定明确的原假设和备选假设。一旦完成，我们可以简单地使用前一节的方法来完成分析。

让我们看看如何在一个例子中实现。

治疗慢性背痛：RCT

成年人的背痛可能非常顽固，难以治疗。常见的方法从皮质类固醇到针灸。随机对照试验（RCT）检验了使用肉毒毒素 A 来治疗的效果。肉毒杆菌毒素是一种神经毒蛋白，会导致肉毒中毒的疾病；维基百科说肉毒杆菌是“已知最致命的毒素”。有七种类型的肉毒杆菌毒素。肉

毒杆菌毒素 A 是可导致人类疾病的类型之一，但也被用于治疗涉及肌肉的各种疾病。福斯特（Foster），克拉普（Clapp）和贾巴里（Jabbari）在 2001 年分析的随机对照试验（RCT）将其用作一种治疗背痛的方法。

将 31 名背痛患者随机分为实验组和对照组，实验组 15 例，对照组 16 例。对照组给予生理盐水，试验是双盲的，医生和病人都不知道他们在哪个组。

研究开始 8 周后，实验组 15 名中的 9 名和对照组 16 名中的 2 名缓解了疼痛（由研究人员精确定义）。这些数据在 `bta` 表中，似乎表明实验有明显的益处。

```
bta = Table.read_table('bta.csv')
bta
```

Group	Result
Control	1
Control	1
Control	0
Control	0
Control	0
Control	0
Control	0
Control	0
Control	0
Control	0

（省略了 21 行）

```
bta.group('Group', np.mean)
```

Group	Result mean
Control	0.125
Treatment	0.6

在实验组中，60% 的患者缓解了疼痛，而对照组只有 12.5%。没有一个患者有任何副作用。

因此，这表示是 A 型肉毒毒素比盐水更好。但结论还没确定。病人随机分配到两组，所以也许差异可能由于偶然？

为了理解这意味着什么，我们必须考虑这样的可能性，即在研究中的 31 个人中，有些人可能比其他人恢复得更好，即使没有任何治疗的帮助。如果他们中的大部分不正常地分配到实验组，只是偶然呢？即使实验组仅仅给予对照组的生理盐水，实验组的结果可能会好于对照组。

为了解释这种可能性，我们首先仔细建立机会模型。

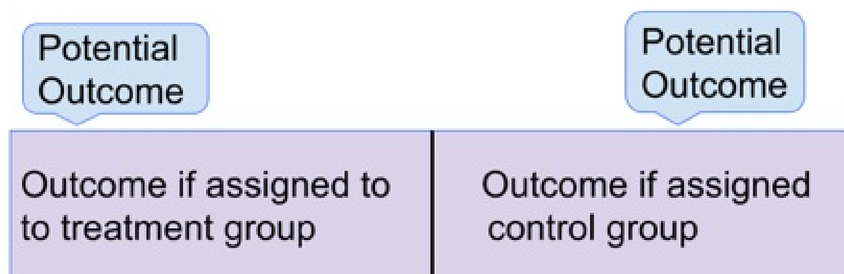
潜在的结果

在患者随机分为两组之前，我们的大脑本能地想象出每个患者的两种可能的结果：患者分配到实验组的结果，以及分配给对照组的结果。这被称为患者的两个潜在的结果。

因此有 31 个潜在的实验结果和 31 个潜在的对照结果。问题关于 31 个结果的这两组的分布。他们是一样的，还是不一样？

我们还不能回答这个问题，因为我们没有看到每个组中的所有 31 个值。我们只能看到随机选择的 16 个潜在的对照结果，以及其余 15 个患者的实验结果。

这是一个展示设定的好方法。每个病人都有一张双面票：



随机化之后，我们可以看到随机选择的一组票的右半部分，以及剩余分组的左半部分。

`observed_outcomes` 表收集每个患者潜在结果的信息，每张“票”的未观察的一半留空。（这只是考虑 `bta` 表的另一种方式，它载有的信息相同。）

```
observed_outcomes = Table.read_table("observed_outcomes.csv")
observed_outcomes.show()
```

Group	Outcome if assigned treatment	Outcome if assigned control
Control	Unknown	1

Control	Unknown	1
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Control	Unknown	0
Treatment	1	Unknown
Treatment	1	Unknown
Treatment	1	Unknown
Treatment	1	Unknown
Treatment	1	Unknown
Treatment	1	Unknown
Treatment	1	Unknown
Treatment	1	Unknown
Treatment	1	Unknown
Treatment	0	Unknown
Treatment	0	Unknown
Treatment	0	Unknown
Treatment	0	Unknown
Treatment	0	Unknown
Treatment	0	Unknown

假设检验

问题是实验是否有用。根据观察得出的结果，问题在于第 2 列（包括未知数）的 31 个“实验”值的分布是否与第 3 列 31 个“对照”值的分布不同（同样包括未知数）。

原假设：所有 31 个潜在“实验”结果的分布与所有 31 个潜在“对照”结果的分布相同。实验与对照没有任何不同。两个样本的差异只是偶然而已。

备选假设：31 个潜在“实验”结果的分布与 31 个对照结果的分布不同。治疗做了一些不同于控制。

为了检验这些假设，请注意，如果原假设是真实的，那么 31 个观察结果的所有分布，对于标记为“对照”的 16 个结果和另外标记为“实验”的 15 个结果将具有相等的可能性。所以我们可以简单地对这些值进行排列，看看这两个组的分布是多么不同。更简单地说，由于数据是数值的，我们可以看到两个均值有多么不同。

这正是我们在上一节中为 A/B 测试所做的。样本 A 现在是对照组，样本 B 是实验组。我们的检验统计量是两组平均值的绝对差。

让我们为均值之间的差异运行我们的排列检验。只有 31 个观测值，所以我们可以运行大量的排列，而不必等待太久的结果。

```
def permutation_test_means(table, variable, classes, repetitions):

    """Test whether two numerical samples
    come from the same underlying distribution,
    using the absolute difference between the means.
    table: name of table containing the sample
    variable: label of column containing the numerical variable
    classes: label of column containing names of the two samples
    repetitions: number of random permutations"""

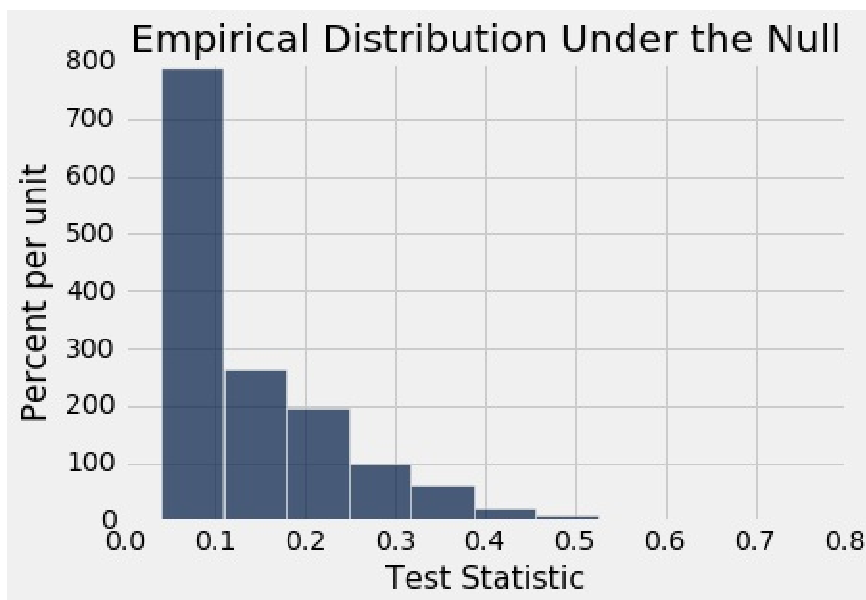
    t = table.select(variable, classes)

    # Find the observed test statistic
    means_table = t.group(classes, np.mean)
    obs_stat = abs(means_table.column(1).item(0) - means_table.column(1).item(1))

    # Assuming the null is true, randomly permute the variable
    # and collect all the generated test statistics
    stats = make_array()
    for i in np.arange(repetitions):
        shuffled_var = t.select(variable).sample(with_replacement=False).column(0)
        shuffled = t.select(classes).with_column('Shuffled Variable', shuffled_var)
        m_tbl = shuffled.group(classes, np.mean)
        new_stat = abs(m_tbl.column(1).item(0) - m_tbl.column(1).item(1))
        stats = np.append(stats, new_stat)

    # Find the empirical P-value:
    emp_p = np.count_nonzero(stats >= obs_stat)/repetitions

    # Draw the empirical histogram of the tvd's generated under the null,
    # and compare with the value observed in the original sample
    Table().with_column('Test Statistic', stats).hist()
    plots.title('Empirical Distribution Under the Null')
    print('Observed statistic:', obs_stat)
    print('Empirical P-value:', emp_p)
permutation_test_means(bta, 'Result', 'Group', 20000)
Observed statistic: 0.475
Empirical P-value: 0.00965
```



经验 P 值非常小（研究报告 P 值为 0.009，这与我们的计算一致），因此证据倾向于备选假设：潜在实验和控制分布是不同的。

这是实验导致差异的证据，因为随机化确保了没有影响结论的混淆变量。

如果实验没有被随机分配，我们的测试仍然会指出我们 31 位患者的实验和背痛结果之间的关联。但要小心：没有随机化，这种关联并不意味着，实验会导致背痛结果的改变。例如，如果患者自己选择是否进行实验，则可能疼痛更严重的患者更可能选择实验，并且甚至在没有药物治疗的情况下，更可能减轻疼痛。预先存在的疼痛将成为分析中的混淆因素。

效果的置信区间

正如我们在上一节中指出的那样，只是简单地得出结论，说治疗是有用的，还不够。我们还想知道它做了什么。

因此，不要对两个基本分布的假设进行是与否的测试，而是仅仅估计它们之间的差异。具体来说，我们查看所有 31 个对照结果的平均值减去所有 31 个实验结果的平均值。这是未知的参数，因为我们只有 16 个对照值和 15 个实验值。

在我们的样本中，平均值的差异是 -47.5%。对照组平均为 12.5%，而治疗组平均为 60%。差异的负面信号表明实验组效果更好。

```
group_means = bta.group('Group', np.mean)
group_means
```

Group	Result mean
Control	0.125
Treatment	0.6

```
group_means.column(1).item(0) - group_means.column(1).item(1)
-0.475
```

但这只是一个样本的结果；样本可能会有所不同。 因此，我们将使用`bootstrap`复制样本，并重新计算差异。 这正是我们在前一节所做的。

```
def bootstrap_ci_means(table, variable, classes, repetitions):

    """Bootstrap approximate 95% confidence interval
    for the difference between the means of the two classes
    in the population"""

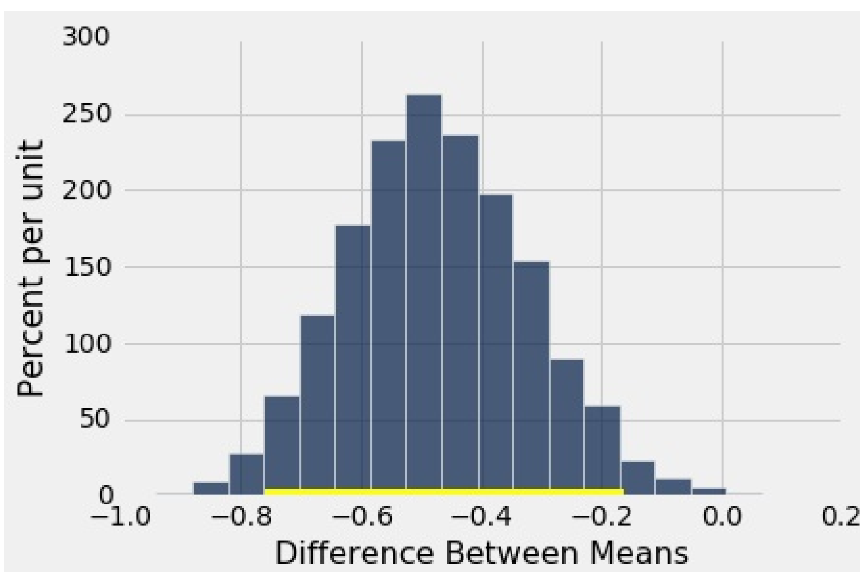
    t = table.select(variable, classes)

    mean_diffs = make_array()
    for i in np.arange(repetitions):
        bootstrap_sample = t.sample()
        m_tbl = bootstrap_sample.group(classes, np.mean)
        new_stat = m_tbl.column(1).item(0) - m_tbl.column(1).item(1)
        mean_diffs = np.append(mean_diffs, new_stat)

    left = percentile(2.5, mean_diffs)
    right = percentile(97.5, mean_diffs)

    # Find the observed test statistic
    means_table = t.group(classes, np.mean)
    obs_stat = means_table.column(1).item(0) - means_table.column(1).item(1)

    Table().with_column('Difference Between Means', mean_diffs).hist(bins=20)
    plots.plot(make_array(left, right), make_array(0, 0), color='yellow', lw=8)
    print('Observed difference between means:', obs_stat)
    print('Approximate 95% CI for the difference between means:')
    print(left, 'to', right)
bootstrap_ci_means(bta, 'Result', 'Group', 20000)
Observed difference between means: -0.475
Approximate 95% CI for the difference between means:
-0.759090909091 to -0.162393162393
```



基本分布的均值之间的差异的约 95% 置信区间，范围是约 -80% 到 -20%。换句话说，实验组好转了 20% 到 80% 左右。

注意这个变化很大的估计。那是因为每个组的样本量只有 15 个左右。虽然这些作用于这些数值而没有进一步的假设，但结果并不十分精确。

元分析

虽然 RCT 确实真名了肉毒杆菌毒素 A 实验帮助了患者，但对 31 名患者进行的研究不足以确定治疗的有效性。这不仅仅是因为样本量小。我们在这一部分的结果对于研究中的 31 位患者是有效的，但我们对所有可能患者的总体真正感兴趣。如果 31 名患者是来自较大总体的随机样本，那么我们的置信区间对该总体是有效的。但他们不是随机样本。

2011 年，一组研究人员对实验的研究进行了元分析。也就是说，他们确定了所有被痛治疗的可用研究，并总结了整理后的结果。

有几项研究，但没有多少可以纳入科学合理的方式："由于非随机性，不完整或未发表的数据，我们排除了 19 项研究的证据。只剩下三个随机对照试验，其中之一是我们在本节研究的。元分析给予它所有研究的最高评价（LBP 代表背痛）："我们确定了三项研究，它们调查了 BoNT 治疗 LBP 的优点，但只有一项的偏差风险低，并且使用非特异性 LBP（N = 31）来评价患者"。

元分析得出的结论是："有一些低质量的证据表明，BoNT 注射剂能改善疼痛，功能，或者两者都比注射生理盐水更好，而且质量很低的证据表明，它比针灸或类固醇注射更好。进一步的研究很可能会对效果评估和我们的信心产生重要影响，未来的试验应该对患者总体，实验方案和比较组进行标准化，争取更多的参与者，并包括长期结果，成本效益分析和临床相关性的发现"。

为了确定医疗有好处，需要很多精心的工作。了解如何分析随机对照试验是这项工作的重要组成部分。现在你们知道了如何实现，你们有条件帮助医疗和其他行业建立因果关系。

十七、更新预测

> 原文：Updating Predictions
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

我们知道如何使用训练数据将一个点划分为两类之一。我们的分类只是对类别的预测，基于最接近我们的新点的，训练点中最常见的类别。

假设我们最终发现了我们的新点的真实类别。然后我们会知道我们的分类是否正确。另外，我们将会有一个新点，可以加入到我们的训练集中，因为我们知道它的类别。这就更新了我们的训练集。所以，我们自然希望，根据新的训练集更新我们的分类器。

本章将介绍一些简单的情况，其中新的数据会使我们更新我们的预测。虽然本章中的例子在计算方面较简单，但是更新方法可以推广到复杂的设定，是机器学习最强大的工具之一。

“更可能”的二分类器

让我们尝试使用数据，将一个点划分为两个类别之一，选择我们认为更可能的类别。为此，我们不仅需要数据，而且还要清楚地描述几率是什么样。

我们将从一个简单的人造情况开始，开发主要的技术，然后跳到更有趣的例子。

假设有个大学班级，其组成如下：

- 60% 的学生为二年级，其余的 40% 是三年级
- 50% 二年级学生已经声明了他们的专业
- 80% 三年级学生已经声明了他们的专业

现在假设我从班上随机挑选一个学生。你能否用“更可能”的标准，将学生划分为二年级或三年级？

你可以，因为这个学生是随机挑选的，所以你知道这个学生是二年级的几率是 60%。这比三年级的 40% 的可能性更大，所以你会把学生划分为二年级。

专业的信息是无关紧要的，因为我们已经知道班上二，三年的比例。

我们有了非常简单的分类器！但是现在假设我给了你一些被挑选的学生的更多信息：

这个学生已经声明了专业。

这个知识会改变你的分类吗？

基于新信息更新预测

现在我们知道学生已经宣布了专业，重要的是要看看年级和专业声明的关系。二年级的学生比三年级多，这仍然正确。但是，三年级的学生，比二年级的学生，声明专业的比例更高，这也是事实。我们的分类器必须考虑到这两个观察。

为了使这个可视化，我们将使用 `students` 表，它包含 100 个学生，每个学生一行，学生的年级和专业比例和数据中相同。

```
students.show(3)
```

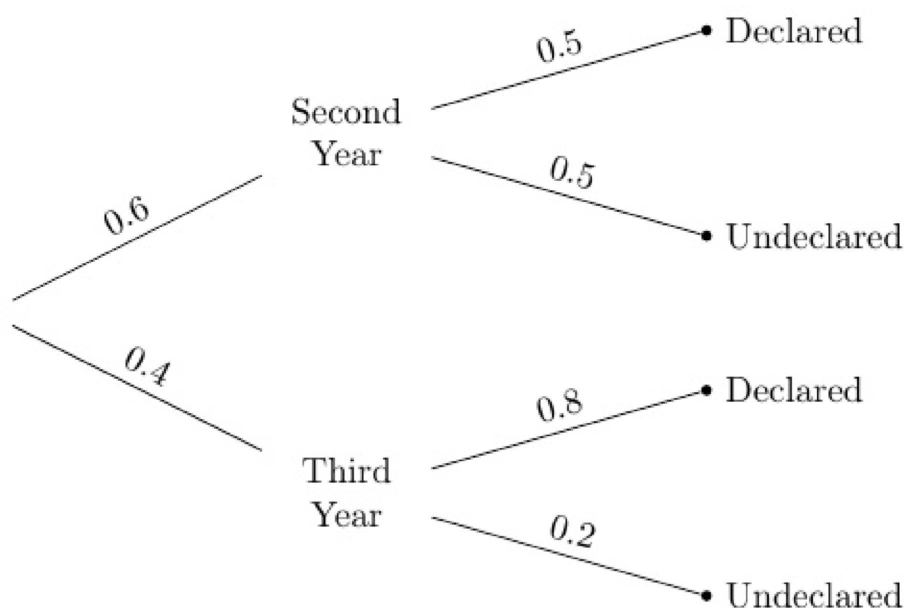
Year	Major
Second	Undeclared
Second	Undeclared
Second	Undeclared

（省略了 97 行）

为了检查比例是否正确，我们使用 `pivot`，按照这两个变量对每个学生进行交叉分类。

```
students.pivot('Major', 'Year')
```

Year	Declared	Undeclared
Second	30	30
Third	32	8



总人数为 100 人，其中二年级 60 人，三年级 40 人。 二年级中，每个专业类别有 50%。 三年级的 40 人中，20% 是未声明的，80% 已声明。 因此，这 100 人的比例和我们问题中的班级相同，我们可以假定，我们的学生是从 100 名学生中随机抽取的。

我们必须选择学生最可能进入的那一行。当我们对这个学生一无所知时，他或她可能在四个单元格中的任何一个，因此更可能在第一行（二年级），因为那里包含更多的学生。

但是现在我们知道这个学生已经声明了专业，所以可能结果的空间已经减少了：现在学生只能在两个已声明的单元格中的一个。

这些单元格共有 62 名学生，其中 32 名是三年级。 这是一半以上，即使不是太多。

所以，考虑到学生专业的新信息，我们必须更新我们的预测，现在将学生划分为三年级。

我们的分类的正确几率是多少？ 对于所有声明了专业的 32 个三年级，我们是正确的，对于那 30 个二年级，我们是错误的。 因此，我们的正确几率大约是 0.516。

换句话说，我们正确几率是声明专业的学生中三年级的比例。

```
32/(30+32)
0.5161290322580645
```

树形图

我们刚刚计算的比例基于 100 名学生。 但是班级没有理由没有 200 名学生，只要单元格中的所有比例都是正确的。 那么我们的计算就变成了 `64 /(60 + 64)`，就是 0.516。

所以计算只取决于不同类别的比例，而不是计数。为了便于比较，比例可以用树形图可视化，直接显示在数据透视表下方。

```
students.pivot('Major', 'Year')
```

Year	Declared	Undeclared
Second	30	30
Third	32	8

像数据透视表一样，该图将学生分成四个不同的组，称为“分支”。请注意，“三年级已声明”分支中的学生比例为 `0.4 x 0.8 = 0.32`，对应于数据透视表中“三年级已声明”单元格中的 32 名学生。“二年级已声明”分支中包含学生的 `0.6 x 0.5 = 0.3`，对应于数据透视表中“二年级已声明”单元格中的 30 个。

我们知道，被挑选的学生属于“已声明”分支。也就是说，学生在两个顶层分支之一。这两个分支现在形成了我们的简化概率空间，所有几率的计算必须相对于这个简化空间的总概率。

所以，考虑到学生已声明专业，他们是三年级的几率可以直接从树中计算出来。答案是相对于两个“已声明”分类的总比例，“三年级已声明”分类的比例。

也就是说，答案是和以前一样，已声明的学生中三年级的比例。

```
(0.4 * 0.8)/(0.6 * 0.5  +  0.4 * 0.8)
0.5161290322580645
```

贝叶斯法则

我们刚刚使用的方法来源于托马斯·贝叶斯牧师（1701-1761）。他的方法解决了所谓的“逆向概率”问题：假设有了新的数据，如何更新之前发现的几率？虽然贝叶斯生活在三个世纪之前，但他的方法现在在机器学习中广泛使用。

我们将在学生总体的背景下讲述这个规则。首先，一些术语：

先验概率。在我们知道所选学生的专业声明状态之前，学生是二年级的几率是 60%，学生是三年级的几率是 40%。这是两个类别的先验概率。

可能性。这是专业状态在给出学生类别情况下的几率；因此可以从树形图中读出。例如，假设学生是二年级，已声明的可能性是 0.5。

后验概率。这些是考虑专业声明状态的信息后，二年级的概率。我们计算了其中的一个：

假设学生已经声明，学生是三年级的后验概率表示为 $P(\text{Third Year} \mid \text{Declared})$，计算如下。

$$P(\text{Third Year} \mid \text{Declared}) = \frac{0.4 \times 0.8}{0.6 \times 0.5 + 0.4 \times 0.8}$$

$$= \frac{(\text{prior probability of Third Year}) \times (\text{likelihood of Declared given Third Year})}{\text{total probability of Declared}}$$

另一个后验概率是：

$$P(\text{Second Year} \mid \text{Declared}) = \frac{0.6 \times 0.5}{0.6 \times 0.5 + 0.4 \times 0.8}$$

$$= \frac{(\text{prior probability of Second Year}) \times (\text{likelihood of Declared given Second Year})}{\text{total probability of Declared}}$$

```
(0.6 * 0.5)/(0.6 * 0.5  +  0.4 * 0.8)
0.4838709677419354
```

这大概是 0.484，还不到一半，与我们三年的分类一致。

请注意，两个后验概率的分母相同：新信息，也就是学生已声明的几率。

正因为如此，贝叶斯方法有时被归纳为比例陈述：

$$\text{posterior} \propto \text{prior} \times \text{likelihood}$$

公式非常便于高效地描述计算。 但是在我们的学生示例这样的情况中，不用公式来思考更简单。 我们仅仅使用树形图。

做出决策

贝叶斯规则的一个主要用途，是基于不完整的信息做出决策，并在新的信息到来时纳入它们。本节指出了在决策时保持你的假设的重要性。

许多疾病的医学检测都会返回阳性或阴性结果。阳性结果意味着，根据检测患者有疾病。阴性结果意味着，检测的结论是患者没有这种疾病。

医学检测经过精心设计，非常准确。但是很少有检测是 100% 准确的。几乎所有检测都有两种错误：

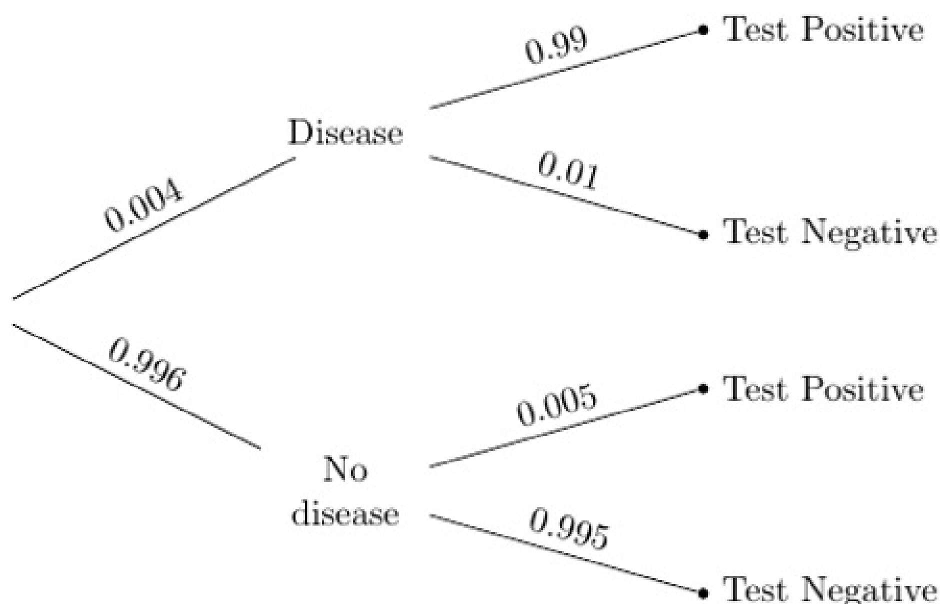
假阳性是，检测结果为阳性，但患者没有该疾病的错误。

假阴性是，检测结果为阴性，但患者确实有这种疾病的错误。

这些错误可能会影响人们的决策。假阳性可能引起焦虑和不必要的治疗（在某些情况下，这是昂贵的或危险的）。如果由于其阴性检测结果，患者未接受治疗，则假阴性可能具有更严重的后果。

罕见疾病的检测

假设总体很大，疾病只占总体的一小部分。 下面的属性图总结了这种疾病的信息，以及它的医学检测。



总的来说，只有千分之四的总体有这种疾病。 检测相当准确：假阳性几率非常小，为 5/1000，但是假阴性更大（尽管还是很小），为 1/100。

个体可能知道也可能不知道他们是否患有这种疾病；通常情况下，人们会进行检测来确认他们是否拥有。

所以假设随机从总体中挑选一个人并进行检测。 如果检测结果是阳性的，你会如何分类：患病还是没有患病？

我们可以通过应用贝叶斯规则，和使用我们的“更可能”的分类器来回答这个问题。 鉴于该人已经检测出阳性，他或她患病的几率是相对于 `Test Positive` 分支中的总比例，顶层分支的比例。

```
(0.004 * 0.99)/(0.004 * 0.99  +  0.996*0.005 )
0.44295302013422816
```

鉴于这个人已经检测出阳性，他或她有这种疾病的几率是大约 44%。 所以我们将它们分类为：没有疾病。

这是一个奇怪的结论。 我们有一个相当准确的检测，一个人检测出阳性，我们的分类是...他们没有这种疾病？ 这似乎没有任何意义。

面对一个令人不安的答案，首先要做的是检查计算。 上面的算法是正确的。 我们来看看是否可以用不同的方式得到相同的答案。

函数 `population` 群体返回 100,000 名患者的结果表格，它的列展示了实际情况和检测结果。检测与树中描述的相同。但是有这种疾病的比例是这个函数的参数。

我们将 0.004 用作参数来调用 `population`，然后调用 `pivot`，对这十万人中的每一个人进行交叉分类。

```
population(0.004).pivot('Test Result', 'True Condition')
```

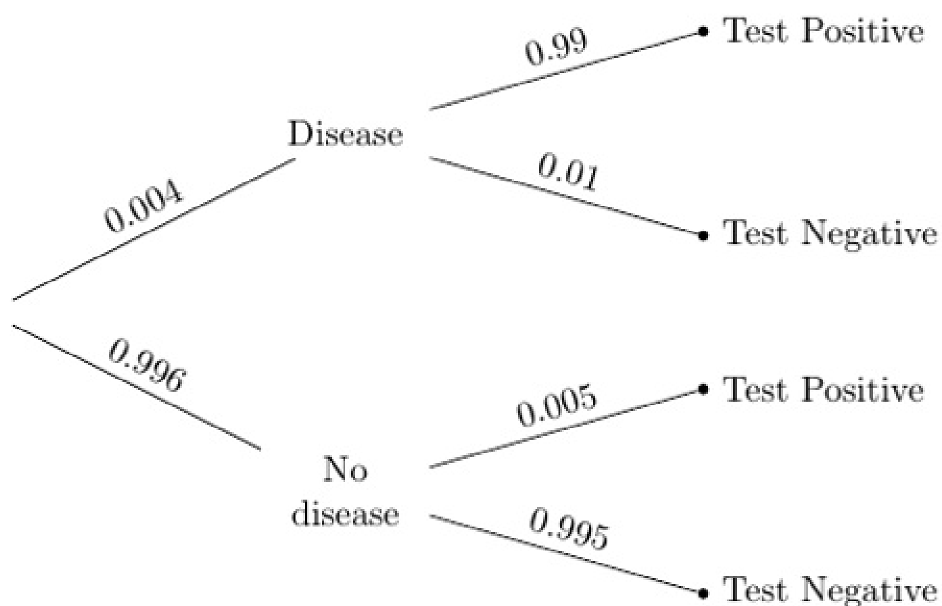
True Condition	Negative	Positive
Disease	4	396
No Disease	99102	498

表的单元格计数正确。例如，根据总体的描述，一千人中有四人患有这种疾病。表格中有十万人，所以 400 人应该有这种病。这就是表格所显示的：`4 + 396 = 400`。在这 400 认中，99% 获得了阳性检测结果：`0.99 x 400 = 396`。

```
396/(396 + 498)
0.4429530201342282
```

这就是我们通过使用贝叶斯规则得到的答案。`Positives` 列中的计数显示为什么它小于 1/2。在阳性的人中，更多的人没有疾病而不是有疾病。

原因是，很大一部分人没有这种疾病。检测出假阳性的一小部分人比眞阳性要多。这在树形图中更容易可视化：



眞阳性的比例是总体一小部分（0.004）的很大一部分（0.99）。假阳性的比例是总体很大一部分（0.996）的一小部分（0.005）。这两个比例是可比的；第二个大一点。

所以，鉴于随机选择的人检测为阳性，我们将他们划分为，更有可能没有疾病，是正确的。

主观先验

正确并不总令人满意。将阳性患者划分为不患有该疾病似乎仍然有些错误，对于这样的精确检测来说。由于计算是正确的，我们来看看我们的概率计算的基础：随机性假设。

我们的假设是，一个随机选择的人进行了检测，并得到了阳性结果。但是这在现实中并没有发生。因为他们认为他们可能有疾病，或者因为他们的医生认为他们可能有疾病，人们去接受检测。被检测的人不是随机选择的总体的成员。

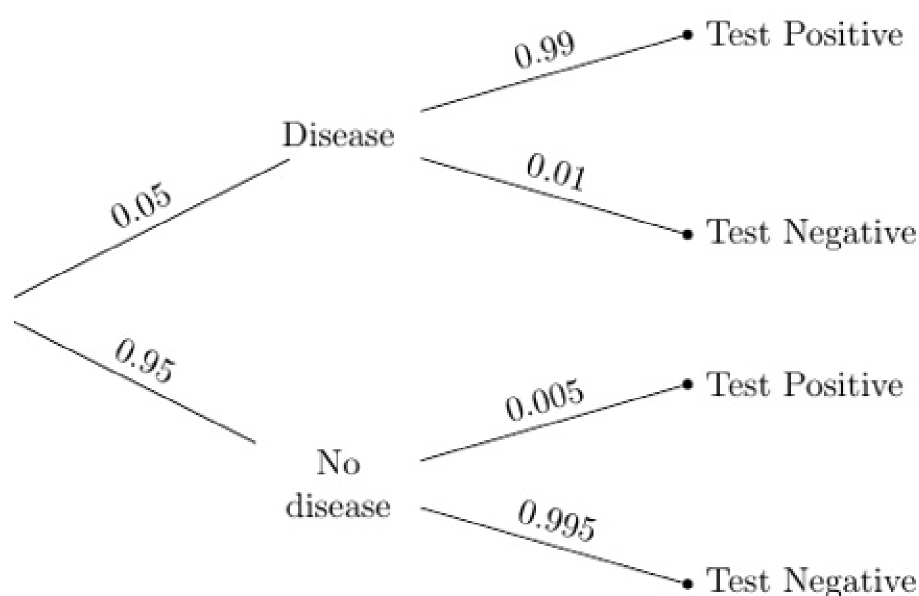
这就是为什么，我们对被检测者的直觉与我们得到的答案不太相符。我们正在想象一个病人接受检测的现实情况，因为有一些理由让他们这样做，而计算基于随机选择的人进行检测。

所以让我们在更现实的假设下重做我们的计算，即病人正在接受检测，因为医生认为病人有发病的机会。

这里需要注意的是，“医生认为有机会”是指医生的意见，而不是总体中的比例。这被称为主观概率。在病人是否患有这种疾病的情况下，这也是主观的先验概率。

一些研究人员坚持认为，所有的概率必须是相对的频率，但主观概率导出都是。候选人赢得下一次选举的几率，大地震在下一个十年将会袭击湾区的几率，某个国家赢得下一届足球世界杯的几率：这些都不是基于相对频率或长期的频率。每个都包含主观因素。涉及它们的所有计算也都有主观因素。

假设医生的主观意见是，患者有 5% 的几率患病。那么树形图中的先验概率将会改变：



鉴于病人检测为阳性，他或她有这种疾病的几率是由贝叶斯规则给出。

```
(0.05 * 0.99)/(0.05 * 0.99  +  0.95 * 0.005)
0.9124423963133641
```

改变先验的效果是惊人的。 即使病人患病的医生的先验概率（5%）很低，一旦患者检测出阳性，患病的后验概率高达 91% 以上。

如果患者检测出阳性，医生认为患者患病是合理的。

确认结果

虽然医生的意见是主观的，但我们可以产生一个人造总体，5% 的人患有这种疾病，并且使用相同的检测来进行检测。 然后，我们可以计算不同类别的人数，看看这些计数是否与我们使用贝叶斯规则得到的答案一致。

我们可以使用 `population(0.05)` 和 `pivot` 构建相应的总体，并看看四个单元格中的计数。

```
population(0.05).pivot('Test Result', 'True Condition')
```

True Condition	Negative	Positive
Disease	50	4950
No Disease	94525	475

在这个人工创建的 10 万人的总体中，有 5000 人（5%）患有这种疾病，其中 99% 的人检测为阳性，导致 4950 人为真阳性。 将其与 475 个假阳性相比：在阳性中，拥有疾病的比例与我们通过贝叶斯规则得到的结果相同。

```
4950/(4950 + 475)
0.9124423963133641
```

因为我们可以一个具有合适比例的总体，我们也可以使用模拟来确认我们的答案是否合理。 `pop_05` 表包含 10 万人的总体，使用医生的先验患病概率 5%，以及检测的错误率来生成。 我们从总体中抽取一个规模为 10,000 的简单随机样本，并提取 `positive` 表，仅包含样本中阳性检测结果的个体。

```
pop_05 = population(0.05)

sample = pop_05.sample(10000, with_replacement=False)

positive = sample.where('Test Result', are.equal_to('Positive'))
```

在这些阳性结果中，真实比例是多少？ 那是拥有这种疾病的阳性的比例：

```
positive.where('True Condition', are.equal_to('Disease')).num_rows/positive.num_rows
0.9131205673758865
```

运行这两个单元格几次，你会发现，阳性中真阳性的比例位于我们通过贝叶斯规则计算的值 0.912 周围。

你也可以以不同参数调用 `population` 函数，来改变先验患病概率，并查看后验概率如何受到影响。